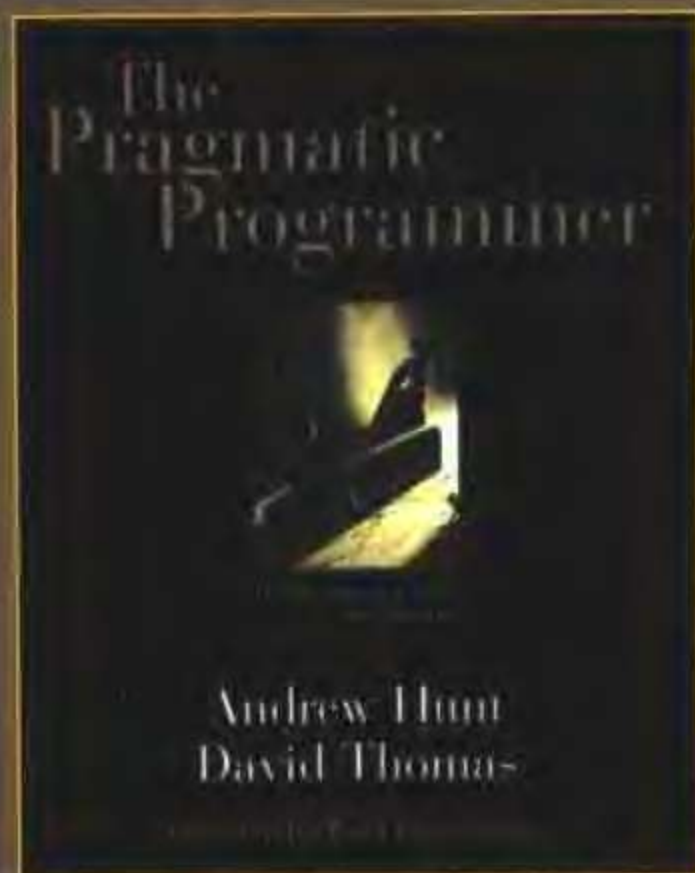


程序员修炼之道

The Pragmatic Programmer

from journeyman to master

从小工到专家



[美] Andrew Hunt 著
David Thomas
马维达 译

The Pragmatic Programmer

程序员修炼之道

from journeyman to master

从小工到专家

“如果我在管理一个项目，这本书的作者就是我想要的人……如果办不到，我就会去读过他们的书的人。”

Ward Cunningham

Andy Hunt 是一位热切的木匠和音乐家。但奇怪的是，人们更需要作为顾问的他。他的工作领域包括电信、银行、金融服务、公共服务，以及一些更奇特的领域：比如医学成像、图形艺术、Internet 服务。Andy 的专长是把经过验证的技术与先进的技术混合在一起，创建各种新颖的——但也是实用的——解决方案。Andy 在北卡罗莱纳州的罗利市拥有自己的顾问公司。

Dave Thomas 喜欢驾驶单引擎飞机飞行，并通过这样的方式为他的习惯付账。为各种难题寻找优雅的解决方案，提供诸多领域里的咨询服务——航空、银行、金融服务、电信、交通运输以及 Internet。在 1994 年移居美国之前，Dave 在英国创立了一家通过了 ISO9001 认证的软件公司，为世界各地的客户开发成熟、定制的软件项目。Dave 现在是一位独立顾问，居住在得克萨斯州的达拉斯。

以 The Pragmatic Programmers, L.L.C 的名义，Dave 与 Andy 正在协同工作，把合起来超过四十年的专业经验带给美国各地的客户。

马维达，《C++ 网络编程（卷 2）》与《ACE 自适应通信环境技术文档》的译者。技术兴趣为 C++ 网络编程（ACE）与分布式对象计算（Internet Communications Engine）。

主页：<http://www.flyingdonkey.com/>

《程序员修炼之道》直接从编程的战场出发，穿过现代软件开发日渐增多的专门化和技术问题，去考察核心的过程——按照需求，编写能工作、可维护、能让用户满意的代码。本书涵盖的主题从个人责任、职业发展，直到用于使你的代码保持灵活，并且易于改编和复用的各种架构技术。阅读本书，你将学会：

- 与软件腐烂作斗争；
- 避开重复知识的陷阱；
- 编写灵活、动态、可适应的代码；
- 防止靠巧合编程；
- 通过合约、断言及异常使你的代码“防弹”；
- 捕捉真正的需求；
- 无情而有效地测试；
- 让你的用户满意；
- 建立注重实效程序员的团队；并且通过自动化使你的开发更严谨。

《程序员修炼之道》由一系列独立的部分组成，讲述了许多富有娱乐性的奇闻轶事、有思想性的例子，以及有趣的类比。本书阐释了软件开发的许多不同方面的最佳实践和重大陷阱。无论你是初学者，是有经验的程序员，还是软件项目经理，只要每天运用这些建议，你很快就会看到你的个人生产率、准确度，以及工作满意度得到提高。你将学习各种技能，发展各种习惯和态度，从而为你的职业生涯的长期成功奠定基础。你将成为一个注重实效的程序员。

ISBN 7-5053-9719-2



9 787505 397194 >



责任编辑：周 筠 方 舟
责任校对：张兴田
封面设计：方 舟

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。
ISBN 7-5053-9719-2 定价：48.00 元

程序员修炼之道

—从小工到专家—

The Pragmatic Programmer

[美] Andrew Hunt David Thomas 著

马维达 译

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 提 要

《程序员修炼之道》由一系列独立的部分组成,涵盖的主题从个人责任、职业发展,直到用于使代码保持灵活、并且易于改编和复用的各种架构技术。利用许多富有娱乐性的奇闻轶事、有思想性的例子以及有趣的类比,全面阐释了软件开发的许多不同方面的最佳实践和重大陷阱。无论你是初学者,是有经验的程序员,还是软件项目经理,本书都适合你阅读。

Simplified Chinese edition copyright © 2004 by PEARSON EDUCATION ASIA LIMITED and Publishing House of Electronics Industry.

The Pragmatic Programmer: From Journeyman to Master, First Edition, ISBN: 0-201-61622-X by Andrew Hunt, David Thomas Copyright © 2000.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书中文简体字翻译版由电子工业出版社和 Pearson Education 培生教育出版亚洲有限公司合作出版。

未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 培生教育出版集团激光防伪标签,无标签者不得销售。

版权贸易合同登记号 图字:01-2003-4993

图书在版编目(CIP)数据

程序员修炼之道:从小工到专家/(美)亨特(Hunt, A.), (美)托马斯(Thomas, D.)著;马维达译.
—北京:电子工业出版社,2004.3

书名原文: The Pragmatic Programmer: From Journeyman to Master

ISBN 7-5053-9719-2

I. 程… II. ①亨… ②托… ③马… III. 程序设计—方法 IV. TP311.11

中国版本图书馆CIP数据核字(2004)第015597号

责任编辑:周 筠 方 舟

责任校对:张兴田

印 刷:北京增富印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编 100036

经 销:各地新华书店

开 本:787×980 1/16 印张:22.5 字数:300千字

印 次:2004年5月第2次印刷

印 数:3000册 定价:48.00元

凡购买电子工业出版社的图书,如有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系。联系电话:(010)68279077。质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dhqq@phei.com.cn。

译 序

本书原名“The Pragmatic Programmer”，也就是“**注重实效的程序员**”。正如书名所示，本书将围绕“注重实效”讲述关于编程的各种话题：个人责任、曳光弹开发、调试策略、元程序设计、按合约设计（Design By Contract）、重构、无情的测试，等等。看到本书的目录，你也许会奇怪，300多页的篇幅，怎么能涵盖如此多内容？但本书的两位作者 Andy Hunt 和 Dave Thomas 的确做到了，他们知道抵达编程的各种维度的途径，并找到了一种言简意赅的方式讲述这些途径；与此同时，在书中还提供了大量资源，可以帮助你找到各种更深入讨论这些话题的读物。本书的各个小节既独立又相关，你可以从头开始阅读，也可以随手翻开任何一页开始阅读——Dave Thomas 就将本书视为一本“洗手间读物”。如果你是编程初学者，你可以从本书中了解到各种编程技术和方法，根据书中的指引拓展你的编程生涯；如果你是富有经验的程序员，同样可以从本书中获益：如果一本书能够全面、明晰地总结你从实践中获得的各种认识、总结你从其他书里散乱地读到的技术和方法，这本书就一定不是无益的。

除了是程序员，Andy Hunt 还是一位木匠和音乐家，而 Dave Thomas 则喜欢驾驶单引擎飞机。尽管作者未曾明言，在本书的许多地方，你都将看到与这样的背景相关的叙述。我想，对于两位作者而言，编程就和木匠活、和音乐创作、或是驾驶飞机一样，既需要禀赋，更需要坚持不懈的学习和训练——这也正是书中所说的，编程是一种技艺，一种需要用心学习的技艺。也许，只有在长久的学习之后，我们才会开始明白书中提到的“hacker”的真正含义：“Someone who loves to program and enjoys being clever about it”（摘自《自由软件杂志》）。

我仍然要感谢侯捷先生和周筠老师，他们像以前一样，为了行业的发展扶掖后进，竭尽心力。谢谢你们的支持和帮助。倘若我未能始终如一，请你们原宥。感谢本书的

编辑方舟先生，他是一个诚恳、好学的年轻人，从不因我的苛刻批评而存有怨言。他的热情、他的年轻，常常让我想起自己那些古怪的、正渐渐没入记忆深处的青春时光。

这是一本“注重实效”的书，其实也可以说，是一本“实用主义”的书。但正因为这样，两位作者在书序的最后给家人的谢辞或许就更加意味深长：

谢谢你们让我们梦想。

马维达 于贵阳

E-mail: weida@flyingdonkey.com

网站: <http://www.flyingdonkey.com>

前言

作为评阅者，我得到了提早阅读你拿在手上的这本书的机会。即使当时还只是草稿，它就已是一本很好的书。**Dave Thomas** 和 **Andy Hunt** 有话要说，并且知道怎样去说。我见过他们所做事情，知道他们所说的将是有效的。我请求让我来撰写这篇前言，以便有机会向你解释其中的原因。

简而言之，本书将告诉你怎样以一种你能够遵循的方式去编程。也许你不认为这是一件困难的事情，但事情却并非如此。为什么？原因之一是，并非所有的编程书籍都是由程序员撰写的。其中有许多是由语言设计者、或是与他们有合作关系的报刊记者编撰而成，意在推销他们的作品。那些书告诉你怎样通过某种编程语言进行表达——这当然很重要，但却只是程序员所做事情的一小部分。

除了通过编程语言进行表达，程序员还要做些什么？嗯，这是一个更深入的问题。大多数程序员在解释他们所做事情这个问题上都会有困难。编程是一项充满了各种细节的工作，追踪这些细节需要专注。时间流逝，代码出现，你查看它们，那里全是些语句。如果你不仔细思考，你也许会以为编程不过就是敲入某种编程语言的语句。你当然错了，但找遍书店的编程专柜，你却还是讲不出所以然。

在《程序员修炼之道》一书中，**Dave** 和 **Andy** 将告诉我们怎样以一种我们能够遵循的方式编程。他们何以能这样聪明？他们不也是和其他程序员一样，专注于各种细节而已吗？答案是他们在做某件事情时，会把注意力投注在他们在做的事情上——然后他们会试着把它做得更好。

设想你在参加一个会议。或许你在想，这个会议没完没了，你还不如去写程序。而 **Dave** 和 **Andy** 会想，他们为什么在开会，他们想知道是否可以通过另外的方式取代会议，并决定是否可使某样事情自动化，以使开会的工作推后。然后他们就会这样去

做。

这就是 Dave 和 Andy 思考的方式。开会并非是某种使他们远离编程的事情。开会就是编程，并且是能够加以改善的编程。我之所以知道他们以这样的方式思考，是因为这是书中的第二条提示：思考你的工作。

那么再设想一下，他们这样思考了几年。很快他们就会拥有一堆解决方案。现在设想他们在工作中使用这些解决方案，又是几年；他们还放弃了其中太过困难、或者不能总是产生结果的解决方案。噢，这样的途径几乎定义了“*pragmatic*”（注重实效）的含义。现在设想他们又用了一两年来写下他们的解决方案。你也许会想，这些信息可真是金矿。你想对了。

两位作者告诉我们他们是怎样编程的，并且是以一种我们能够遵循的方式来告诉我们的。但这一陈述的后半部分的含义也许要多于你所想到的。让我来解释一下。

作者一直在小心避免提出软件开发理论。这是一件幸运的事情，因为如果他们那样做了，他们就不得不为了捍卫他们的理论而对各章进行“调整”。这样的“调整”是，比如说，物理科学中的传统，在这些学科中，理论不是最终成为定律，就是被静静地丢弃。而另一方面，编程所具有的法则（如果有）却非常少。所以围绕想要成为法则的东西形成的编程建议在纸面上也许显得很好，而在实践中却无法让人满意。这也是那么多方法学书籍误入歧途之处。

我研究这一问题已有十多年，并发现一种叫做模式语言（*pattern language*）的方法最有前途。简而言之，模式就是解决方案，而模式语言就是相互支援的若干解决方案的系统。围绕着对这些系统的探求，已经形成了一整个社群。

本书不只是一堆提示。它是一种“披着羊皮”的模式语言。我这样说，是因为每一条提示都汲取自经验、作为具体建议讲授、并与其他提示关联而形成系统，是这些特征使我们能够学习并遵循模式语言。在本书中它们以同样的方式发挥着作用。

你可以遵循本书的建议，因为它们是具体的。你不会发现含混不清的抽象。Dave 和 Andy 直接为你而写，就好像每一条提示都是能给你的编程生涯供给能量的重大策略。他们让提示保持简单，他们讲故事，他们使用轻松的笔触，他们接着还给出了各

种问题的解答，这些问题将在你进行尝试时出现。

不仅如此。在你阅读了十或十五条提示之后，你将开始看到工作的另外一个维度。我们有时称之为“*QWAN*”，也即“*quality without a name*”（无名的品质）。本书的哲学将渗入你的意识，并与你自己的哲学交融在一起。它不鼓吹，它只是讲述什么可行。但在讲述中却又有更多的东西到临。这正是本书美之所在：它体现它的哲学，以如此谦逊的方式。

这就是它：一本易于阅读——也易于应用——的关于整个编程实践的书。我一直在不断讲述它为何有效，而你关心的也许只是它的确有效。它的确有效，你会看到的。

——Ward Cunningham

序

本书将帮助你成为更好的程序员

不论你是单独的开发者，是大型项目团队中的一员，还是同时与许多客户共事的顾问，这都没有关系。本书将帮助你，作为一个个体，更好地完成工作。本书不是理论书籍——我们专注于实践性的话题，专注于让你利用你的经验做出更有见识的决策。*pragmatic* 一词来自拉丁语的 *pragmaticus*——“精于事务”——后者又源自希腊语的 *πραγματις*，意为“to do”。这是一本关于“doing”的书。

编程是一种技艺。用最简单的话表述，编程可归结为让计算机做你（或你的用户）想要它做的事情。作为程序员，你既是倾听者，又是顾问；既是解释者，又是发号施令者。你设法捕捉难以捉摸的需求，并找到表达它们的方式，让一台纯粹的机器能够合理地处理它们。你设法为你的工作建立文档，以使他人能够理解它；你还设法使你的工作工程化，以使他人能够以它为基础进行构建。还有，你设法在项目时钟无休止的“嘀嗒”声的催迫下完成所有这些工作。你每天都在创造小小的奇迹。

编程是艰难的工作。

有许多人声称要给你帮助。工具供应商吹嘘它们的产品所展现出的奇迹。方法学古鲁（guru）允诺说他们的技术保证有效。每个人都声称他们的编程语言是最好的，而每一种操作系统都是对所有可以想象得到的问题的解答。

当然，所有这些都不是真的。并不存在容易的答案。也不存在最佳解决方案这样东西，无论它是工具，是语言，还是操作系统。能够存在的只是在某些特定情形下更为适宜的系统。

这正是注重实效（pragmatism）登场的地方。你不应该局限于任何特定的技术，而是应该拥有足够广博的背景和经验基础，以让你能在特定情况下选择好的解决方案。

你的背景源自对计算机科学的基本原理的理解，而你的经验来自广泛的实际项目。理论与实践的结合使你强大起来。

你调整你的方法，以适应当前情形与环境。你判断对项目有影响的所有因素的相对重要性，并利用你的经验制定适宜的解决方案。你随着工作的进展持续不断地进行这样的活动。**注重实效的程序员**不仅要完成工作，而且要完成得漂亮。

谁应该阅读本书

本书的目标读者是想要变得更为有效、更多产的程序员。或许你觉得灰心，因为你好像没有在实现自己的潜能。或许你看见同事似乎在使用一些工具，使他们自己比你更多产。也许你现在的工作使用的是较老的技术，而你却想要知道怎样把较新的思想应用于你正在做的工作。

我们不会假装自己拥有所有的（或者即使是大部分）答案，我们的思想也并非适用于所有情况。我们所能说的只是，如果你遵循我们的方法，你将迅速地获取经验，你的生产力将会提高，并且你还将更好地理解整个开发过程。你将能编写更好的软件。

注重实效的程序员有哪些特征

每一个开发者都是独特的，有着个人的力量和弱点、偏好和嫌恶。随着时间的过去，每一个开发者都会营造出他或她自己的个人环境。这个环境会有力地反映这个程序员的个性，就像他或她的业余爱好、衣着或是发型一样。但是，如果你是一个**注重实效的程序员**，你就会具有下列特征中的许多特征：

- **早期的采纳者/快速的改编者。**你具有技术和技巧上的直觉，你喜爱试验各种事物。给你一样新东西，你很快就能把握它，并把它与你的知识的其余部分结合在一起。你的自信出自经验。

- **好奇。**你喜欢提问。那很漂亮——你是怎么做的？你用那个库时有问题吗？我听说的这个 BeOS 是什么？符号链接是怎样实现的？你是收集小知识的林鼠（pack rat），每一条小知识都可能会影响今后几年里的某项决策。
- **批判的思考者。**你不会不首先抓住事实而照搬别人的说法。当同事说“因为就该那么做”或者供应商允诺为你的全部问题提供解决方案时，你就会嗅到挑战的气息。
- **有现实感。**你会设法理解你面临的每个问题的内在本质。这样的现实主义给了你良好的感知能力：事情有多困难，需要多长时间？让你自己了解某个过程会有困难，或是要用一点时间才能完成，能够给予你坚持不懈的毅力。
- **多才多艺。**你尽力熟悉广泛的技术和环境，并且努力工作，以与各种新发展并肩前行。尽管你现在的工作也许只要求你成为某方面的专才，你却总是能够转向新的领域和新的挑战。

我们把最基本的特征留到了最后。所有**注重实效的程序员**都具有这些特征。它们基本得足以用提示的方式来陈述：

提示 1

Care About Your Craft
关心你的技艺

我们觉得，除非你在乎能否漂亮地开发出软件，否则其他事情都是没有意义的。

提示 2

Think! About Your Work
思考！你的工作

为了让你成为**注重实效的程序员**，我们向你发出挑战：在你做某件事情的时候思

考你在做什么。这不是对当前实践的一次性审计——它是对你每一天、在每一次开发上所做出的每一项决策的批判评估。不要依靠自动驾驶仪。不间断地思考，实时地批判你的工作。老IBM公司的箴言，THINK!，是**注重实效的程序员**的曼特罗（mantra，印度教或佛教的颂歌、咒语——译注）。

如果这在你听来是困难的工作，那么你就正在展现出有现实感的特征。这将占据你的一些宝贵时间——很可能是已经处在极大压力之下的时间。酬劳则是更为活跃地参与你喜爱的工作、感觉到自己在掌握范围日增的各种主题以及因感受到持续的进步而欢愉。从长远来说，你在时间上的投入将会随着你和你的团队变得更为高效、编写出更易于维护的代码以及开会时间的减少而得到回报。

注重实效的个体，大型的团队

有人觉得在大型团队或复杂项目中没有个性的位置。“软件构造是工程学科。”他们说：“如果个别的团队成员自行其是，团队就会崩溃。”

我们不同意这种看法。

软件的构造应该是工程学科。但是，这并不排斥个人的技艺。想一想中世纪在欧洲建造的大教堂，每一座都需要数千人年的努力，跨越许多个十年。学到的教训被传递给下一批建造者，后者又通过他们的造诣去提高结构工程的水平。但木匠、石匠、雕刻工和玻璃工都是手艺人，他们解释各种工程需求，以制造超越了建筑的纯粹机械方面的一个整体。他们相信，他们个人的贡献支撑了整个项目：

我们，采集的只是石头，却必须时刻展望未来的大教堂。

——采石工人的信条

在一个项目的总体结构中，总有个性和技艺的位置。就软件工程目前的状态而言，事情就更是如此。一百年之后，我们的工程看起来或许已很古老，就像是中世纪的大教堂建造者所使用的技术在今天的土木工程师看来很古老一样，但我们的技艺却仍将

受到尊重

它是一个持续的过程

一位参观英格兰伊顿公学的游客问那里的园丁，他是怎样让草坪变得如此完美的。“那很容易，”园丁回答说，“你只要每天早晨拂去露水，每隔一天刈一次草，每个星期碾压一次就行了。”

“就是这些吗？”游客问。

“就是这些，”园丁回答说，“这样做上 500 年，你也将拥有一片漂亮的草坪。”

了不起的草坪需要每天给予一点关心，了不起的程序员也是这样。管理顾问们喜欢在谈话中扔出“*kaizen*”这个词。“Kaizen”是一个日语术语（译注：*kaizen*，日文“改善(かいぜん)”），表达的是持续地做出许多小改进的概念。它被认为是日本制造业在生产率与质量方面取得长足进步的主要原因之一，并且在世界各地得到了广泛的效仿。Kaizen 也适用于个人。每天为提炼你所拥有的技能而工作，为把新的工具增加到你的技能列表中而工作。与伊顿的草坪不同，你在几天之中就将开始看到结果。几年之后，你将会惊奇你的经验得到了怎样的发展，你的技能得到了怎样的提升。

本书的组织方式

本书由一系列小节组成。每一节都是独立的，并且讨论一个特定的话题。你会发现大量交叉引用，帮助把各个话题置入相关语境（context）中。请随意以任何次序阅读各节——这不是一本需要你从头到尾顺次阅读的书。

有时你会遇到标有提示 nn（比如第 xix 页上的提示 1：“关心你的技艺”）的。除了强调文本中的要点以外，我们还觉得提示有其自身的生命——我们每天都和它们生活在一起。在本书末尾你可以找到全部提示的一览表。

附录 A 包含了一组资源：本书的参考文献、Web 资源的 URL 列表、以及我们推荐的期刊、书籍和专业组织的列表。贯穿全书你将会发现对参考文献和 URL 列表（比如[KP99]和[URL18]）的分别引用。

我们还在适当的地方包括了一些练习和挑战。练习通常有相对直接的答案，而挑战则更为开放。为了让你对我们的想法有所了解，我们还在附录 B 中包括了我们对练习的解答，但这些练习很少只有一个正确的解决方案。挑战可以用做小组讨论的基础，或是高级编程课程中的论文作业。

名称的内涵

“我使用词语时，” Humpty Dumpty 用一种轻蔑的语调说，“我要它是什么意思它就是什么意思——不多也不少。”

——Lewis Carroll, *Through the Looking-Glass*

在全书的各个地方，你会遇到各种各样的行话（jargon）——它们或者是非常纯正的英语，被故意误用来表示某些技术事物；或者是人为捏造的可怕词语，由对语言怀有嫉妒之心的计算机科学家赋予了各种含义。我们第一次使用某个这样的行话词语时，会试着定义它，或至少是给出关于其含义的提示。但是，我们确信某些行话已经掉进了故纸堆，而另一些，比如对象和关系数据库，十分常用，加上定义反而让人厌烦。如果你确实遇到了一个你未曾见过的术语，请不要轻易跳过它。花点时间查一查，或是在网上，或是在某本计算机科学课本中。而且，如果有机会，就给我们发个邮件，抱怨一下，让我们在下版中加上定义。

说过所有这些之后，我们决定报复一下计算机科学家们。有时候，有一些行话词语能够完好地表达各种概念，我们却决定忽略它们。为什么？因为已有的行话通常都被限定在特定的问题领域中，或是特定的开发阶段。而这本书的基本哲学之一就是推荐的大多数技术都是普遍适用的：例如，模块性适用于代码、设计、文档以及团队组织。当我们想要在更宽泛的语境中使用传统的行话词语时，它会造成混淆——

我们似乎无法克服原来的术语所附带的包袱。在发生这样的事情时，我们就会发明我们自己的术语，并以此为语言的衰败做出贡献。

源码与其他资源

书中所示的大部分代码都摘录自可从我们的网站上下载的可编译源文件：

`www.pragmaticprogrammer.com`

你也能在那里找到我们认为有用的资源链接，以及本书的更新及关于其他**注重实效的程序员**的开发活动的新闻。

给我们发送反馈

我们将重视你的来信。意见、建议、文本中的错误、或是例子中的问题都很欢迎。我们的邮件地址是：

`ppbook@pragmaticprogrammer.com`

致谢

当我们开始撰写本书时，我们并不知道它最终会成为这样一个协作的成果。

Addison-Wesley 一直都很卓越，从选题到待印制的拷贝，他们领着一对幼稚的黑客走过了整个书籍制作过程。十分感谢 **John Wait** 和 **Meera Ravindiran** 最初的支持；感谢 **Mike Hendrickson**，我们热心的编辑（和小气的封面设计者！），还有 **Lorraine Ferrier** 以及 **John Fuller** 在制作上提供的帮助；感谢不屈不挠的 **Julie Debaggis** 把我们大家团结在一起。

然后是评阅者：**Greg Andress**、**Mark Cheers**、**Chris Cleeland**、**Alistair Cockburn**、**Ward Cunningham**、**Martin Fowler**、**Thanh T. Giang**、**Robert L. Glass**、**Scott**

Henninger、Michael Hunter、Brian Kirby、John Lakos、Pete McBreen、Carey P. Morris、Jared Richardson、Kevin Ruland、Eric Starr、Eric Vought 和 Chris Van Wyk 没有他们细心的评阅和宝贵的洞见，这本书不会像现在这样易读、准确，并且会加长一倍。谢谢你们大家的时间和智慧。

几年来，我们与许多锐意进取的客户工作在一起，取得并提炼了我们在此写下的经验。最近，我们有幸与 **Peter Gehrke** 一道参与了若干大型项目的开发。非常感谢他对我们的技术的支持和热情。

本书通过 Linux 下的 Bash 和 zsh shell，使用 LaTeX、pic、Perl、dvips、ghostview、ispell、GNU make、CVS、Emacs、XEmacs、EGCS、GCC、Java、iContract 和 SmallEiffel 制作。令人惊讶的是，所有这些奇妙的软件都可以自由获取。我们应该向世界各地的**注重实效的程序员**大声说：“谢谢你们”，他们为我们大家奉献了上述的以及其他的作品。我们尤其要感谢 **Reto Kramer** 在 iContract 方面给我们的帮助。

最后，但绝非最不重要的是，我们对我们的家人亏欠甚多。她们不仅要忍受深夜的键盘敲击声、巨额的电话账单以及我们长期心不在焉的状态，她们还一次又一次很有风度地阅读了我们所写下的东西。谢谢你们让我们梦想。

Andy Hunt
Dave Thomas

目 录

译序	xi
前言	xiii
序	xvii
第 1 章 注重实效的哲学.....	1
1 我的源码让猫给吃了.....	2
2 软件的熵.....	4
3 石头汤与煮青蛙.....	7
4 足够好的软件.....	9
5 你的知识资产.....	12
6 交流!	18
第 2 章 注重实效的途径.....	25
7 重复的危害.....	26
8 正交性.....	34
9 可撤销性.....	44
10 曳光弹.....	48
11 原型与便笺.....	53
12 领域语言.....	57
13 估算.....	64
第 3 章 基本工具.....	71

14 纯文本的威力.....	73
15 shell 游戏.....	77
16 强力编辑.....	82
17 源码控制.....	86
18 调试.....	90
19 文本操纵.....	99
20 代码生成器.....	102
第 4 章 注重实效的偏执.....	107
21 按合约设计.....	109
22 死程序不说谎.....	120
23 断言式编程.....	122
24 何时使用异常.....	125
25 怎样配平资源.....	129
第 5 章 弯曲, 或折断.....	137
26 解耦与得墨忒耳法则.....	138
27 元程序设计.....	144
28 时间耦合.....	150
29 它只是视图.....	157
30 黑板.....	165
第 6 章 当你编码时.....	171
31 靠巧合编程.....	172
32 算法速率.....	177
33 重构.....	184
34 易于测试的代码.....	189

35 邪恶的向导.....	198
第 7 章 在项目开始之前.....	201
36 需求之坑.....	202
37 解开不可能解开的谜题.....	212
38 等你准备好.....	215
39 规范陷阱.....	217
40 圆圈与箭头.....	220
第 8 章 注重实效的项目.....	223
41 注重实效的团队.....	224
42 无处不在的自动化.....	230
43 无情的测试.....	237
44 全都是写.....	248
45 极大的期望.....	255
46 傲慢与偏见.....	258
附录 A 资源.....	261
专业协会.....	262
建设藏书库.....	262
Internet 资源	266
参考文献.....	275
附录 B 练习解答.....	279
索引	309
注重实效的程序员之快速参考指南.....	323

第 1 章

注重实效的哲学 A Pragmatic Philosophy

注重实效的程序员的特征是什么？我们觉得是他们处理问题、寻求解决方案时的态度、风格、哲学。他们能够越出直接的问题去思考，总是设法把问题放在更大的语境中，总是设法注意更大的图景。毕竟，没有这样的更大的语境，你又怎能注重实效？你又怎能做出明智的妥协和有见识的决策？

他们成功的另一关键是他们对他们所做的每件事情负责，关于这一点，我们将在“我的源码让猫给吃了”中加以讨论。因为负责，**注重实效的程序员**不会坐视他们的项目土崩瓦解。在“软件的熵”中，我们将告诉你怎样使你的项目保持整洁。

大多数人发现自己很难接受变化，有时是出于好的理由，有时只是因为固有的惰性。在“石头汤与煮青蛙”中，我们将考察一种促成变化的策略，并（出于对平衡的兴趣）讲述一个忽视渐变危险的两栖动物的警世传说。

理解你的工作的语境的好处之一是，了解你的软件必须有多好变得更容易了。有时接近完美是惟一的选择，但常常会涉及各种权衡。我们将在“足够好的软件”中探究这一问题。

当然，你需要拥有广泛的知识 and 经验基础才能赢得这一切。学习是一个持续不断的过程。在“你的知识资产”中，我们将讨论一些策略，让你“开足马力”。

最后，我们没有人生活在真空中。我们都要花大量时间与他人打交道。在“交流！”中列出了能让我们更好地做到这一点的几种途径。

注重实效的编程源于注重实效的哲学的哲学。本章将为这种哲学设立基础。

1 我的源码让猫给吃了

在所有弱点中，最大的弱点就是害怕暴露弱点。

——J. B. Bossuet, Politics from Holy Writ, 1709

依据你的职业发展、你的项目和你每天的工作，为你自己和你的行为负责这样一种观念，是注重实效的哲学的一块基石。**注重实效的程序员**对他或她自己的职业生涯负责，并且不害怕承认无知或错误。这肯定并非是编程最令人愉悦的方面，但它肯定会发生——即使是在最好的项目中。尽管有彻底的测试、良好的文档以及足够的自动化，事情还是会出错。交付晚了，出现了未曾预见到的技术问题。

发生这样的事情，我们要设法尽可能职业地处理它们。这意味着诚实和坦率。我们可以为我们的能力自豪，但对于我们的缺点——还有我们的无知和我们的错误——我们必须诚实。

负责

责任是你主动担负的东西。你承诺确保某件事情正确完成，但你不一定能直接控制事情的每一个方面。除了尽你所能以外，你必须分析风险是否超出了你的控制。对于不可能做到的事情或是风险太大的事情，你有权不去为之负责。你必须基于你自己的道德准则和判断来做出决定。

如果你确实同意要为某个结果负责，你就应切实负起责任。当你犯错误（就如同我们所有人都会犯错误一样）、或是判断失误时，诚实地承认它，并设法给出各种选择。不要责备别人或别的东西，或是拼凑借口。不要把所有问题都归咎于供应商、编程语言、管理部门、或是你的同事。也许他（它）们全体或是某几方在其中扮演了某种角

色，但你可以选择提供解决方案，而非寻找借口。

如果存在供应商不能按时供货的风险，你应该预先制定一份应急计划。如果磁盘垮了——带走了你的所有源码——而你没有做备份，那是你的错。告诉你的老板“我的源码让猫给吃了”也无法改变这一点。

提示 3

Provide Options, Don't Make Lame Excuses

提供各种选择，不要找蹩脚的借口

在你走向任何人、告诉他们为何某事做不到、为何耽搁、为何出问题之前，先停下来，听一听你心里的声音。与你的显示器上的橡皮鸭交谈，或是与猫交谈。你的辩解听起来合理，还是愚蠢？在你老板听来又是怎样？

在你的头脑里把谈话预演一遍。其他人可能会说什么？他们是否会问：“你试了这个吗……”，或是“你没有考虑那个吗？”你将怎样回答？在你去告诉他们坏消息之前，是否还有其他你可以再试一试的办法？有时，你其实知道他们会说什么，所以还是不要给他们添麻烦吧。

要提供各种选择，而不是找借口。不要说事情做不到；要说明能够做什么来挽回局面。必须把代码扔掉？给他们讲授重构的价值（参见重构，184 页）。你要花时间建立原型（prototyping），以确定最好的继续前进的方式（参见原型与便笺，53 页）？你要引入更好的测试（参见易于测试的代码，189 页；以及无情的测试，237 页）或自动化（参见无处不在的自动化，230 页），以防止问题再度发生？又或许你需要额外的资源。不要害怕提出要求，也不要害怕承认你需要帮助。

在你大声说出它们之前，先设法把蹩脚的借口清除出去。如果你必须说，就先对你的猫说。反正，如果小蒂德尔丝（Tiddles，BBC 在 1969–1974 年播出的喜剧节目“Monty Python's Flying Circus”中的著名小母猫——译注）要承受指责……

相关内容：

- 原型与便笺，53 页
- 重构，184 页
- 易于测试的代码，189 页
- 无处不在的自动化，230 页
- 无情的测试，237 页

挑战

- 如果有人——比如银行柜台职员、汽车修理工或是店员——对你说蹩脚的借口，你会怎样反应？结果你会怎样想他们和他们的公司？

2 软件的熵

尽管软件开发几乎不受任何物理定律的约束，熵（entropy）对我们的影响却很大。熵是一个来自物理学的概念，指的是某个系统中的“无序”的总量。遗憾的是，热力学定律保证了宇宙中的熵倾向于最大化。当软件中的无序增长时，程序员们称之为“软件腐烂”（software rot）。

有许多因素可以促生软件腐烂。其中最重要的一个似乎是开发项目时的心理（或文化）。即使你的团队只有你一个人，你开发项目时的心理也可能是非常微妙的事情。尽管制定了最好的计划，拥有最好的开发者，项目在其生命期中仍可能遭遇毁灭和衰败。而另外有一些项目，尽管遇到巨大的困难和接连而来的挫折，却成功地击败自然的无序倾向，设法取得了相当好的结果。

是什么造成了这样的差异？

在市区，有些建筑漂亮而整洁，而另一些却是破败不堪的“废弃船只”。为什么？犯罪和城市衰退领域的研究者发现了一种迷人的触发机制，一种能够很快将整洁、完整和有人居住的建筑变为破败的废弃物的机制[WK82]

破窗户

一扇破窗户，只要有那么一段时间不修理，就会渐渐给建筑的居民带来一种废弃感——一种职权部门不关心这座建筑的感觉。于是又一扇窗户破了。人们开始乱扔垃圾。出现了乱涂乱画。严重的结构损坏开始了。在相对较短的一段时间里，建筑就被损毁得超出了业主愿意修理的程度，而废弃感变成了现实。

“破窗户理论”启发了纽约和其他大城市的警察部门，他们对一些轻微的案件严加处理，以防止大案的发生。这起了作用：管束破窗户、乱涂乱画和其他轻微违法事件减少了严重罪案的发生。

提示 4

Don't Live with Broken Windows

不要容忍破窗户

不要留着“破窗户”（低劣的设计、错误决策、或是糟糕的代码）不修。发现一个就修一个。如果没有足够的时间进行适当的修理，就用木板把它钉起来。或许你可以把出问题的代码放入注释（comment out），或是显示“未实现”消息，或是用虚设的数据（dummy data）加以替代。采取某种行动防止进一步的损坏，并说明情势处在你的控制之下。

我们看到过整洁、运行良好的系统，一旦窗户开始破裂，就相当迅速地恶化。还有其他一些因素能够促生软件腐烂，我们将在别处探讨它们，但与其他任何因素相比，置之不理都会更快地加速腐烂的进程。

你也许在想，没有人有时间到处清理项目的所有碎玻璃。如果你继续这么想，你就最好计划找一个大型垃圾罐，或是搬到别处去。不要让熵赢得胜利。

灭火

作为对照，让我们讲述 Andy 的一个熟人的故事。他是一个富得让人讨厌的富翁，拥有一所完美、漂亮的房子，里面满是无价的古董、艺术品，以及诸如此类的东西。有一天，一幅挂毯挂得离他的卧室壁炉太近了一点，着了火。消防人员冲进来救火——和

他的房子。但他们拖着粗大、肮脏的消防水管冲到房间门口却停住了——火在咆哮——他们要在前门和着火处之间铺上垫子。

他们不想弄脏地毯

这的确是一个极端的事例，但我们必须以这样的方式对待软件。一扇破窗户——一段设计低劣的代码、团队必须在整个项目开发过程中加以忍受的一项糟糕的管理决策——就足以使项目开始衰败。如果你发现自己在有好些破窗户的项目里工作，会很容易产生这样的想法：“这些代码的其余部分也是垃圾，我只要照着做就行了。”项目在这之前是否一直很好，并没有什么关系。在最初得出“破窗户理论”的一项实验中，一辆废弃的轿车放了一个星期，无人理睬，而一旦有一扇窗户被打破，数小时之内车上的设备就被抢夺一空，车也被翻了个底朝天。

按照同样的道理，如果你发现你所在团队和项目的代码十分漂亮——编写整洁、设计良好，并且很优雅——你就很可能会格外注意不去把它弄脏，就和那些消防员一样。即使有火在咆哮（最后期限、发布日期、会展演示，等等），你也不会想成为第一个弄脏东西的人。

相关内容：

- 石头汤与煮青蛙，7 页
- 重构，184 页
- 注重实效的团队，224 页

挑战

- 通过调查你周边的计算“环境”，帮助增强你的团队的能力。选择两或三扇“破窗户”，并与你的同事讨论问题何在，以及怎样修理它们。
- 你能否说出某扇窗户是何时破的？你的反应是什么？如果它是他人的决策所致，或者是管理部门的指示，你能做些什么？

3 石头汤与煮青蛙

三个士兵从战场返回家乡，在路上饿了。他们看见前面有村庄，就来了精神——他们相信村民会给他们一顿饭吃。但当他们到达那里，却发现门锁着，窗户也关着。经历了多年战乱，村民们粮食匮乏，并把他们有的一点粮食藏了起来。

士兵们并未气馁，他们煮开一锅水，小心地把三块石头放进去。吃惊的村民们走出来望着他们。

“这是石头汤。”士兵们解释说。“就放石头吗？”村民们问。“一点没错——但有人说加一些胡萝卜味道更好……”一个村民跑开了，又很快带着他储藏的一篮胡萝卜跑回来。

几分钟之后，村民们又问：“就是这些了吗？”

“哦，”士兵们说：“几个土豆会让汤更实在。”又一个村民跑开了。

接下来的一小时，士兵们列举了更多让汤更鲜美的配料：牛肉、韭菜、盐，还有香菜。每次都会有一个不同的村民跑回去搜寻自己的私人储藏品。

最后他们煮出了一大锅热气腾腾的汤。士兵们拿掉石头，和所有村民一起享用了一顿美餐，这是几个月以来他们所有人第一次吃饱饭。

在石头汤的故事里有两层寓意。士兵戏弄了村民，他们利用村民的好奇，从他们那里弄到了食物。但更重要的是，士兵充当催化剂，把村民团结起来，和他们一起做到了他们自己本来做不到的事情——一项协作的成果。最后每个人都是赢家。

你常常也可以效仿这些士兵。

在有些情况下，你也许确切地知道需要做什么，以及怎样去做。整个系统就在你的眼前——你知道它是对的。但请求许可去处理整个事情，你会遇到拖延和漠然。大家要设立委员会，预算需要批准，事情会变得复杂化。每个人都会护卫他们自己的资源。有时候，这叫做“启动杂役”（start-up fatigue）。

这正是拿出石头的时候。设计出你可以合理要求的东西，好好开发它。一旦完成，就拿给大家看，让他们大吃一惊。然后说：“要是我们增加……可能就会更好。”假装那并不重要。坐回椅子上，等着他们开始要你增加你本来就想要的功能。人们发现，参与正在发生的成功要更容易。让他们瞥见未来，你就能让他们聚集在你周围¹。

提示 5**Be a Catalyst for Change**

做变化的催化剂

村民的角度

另一方面，石头汤的故事也是关于温和而渐进的欺骗的故事。它讲述的是过于集中的注意力。村民想着石头，忘了世界的其余部分。我们都是这样，每一天，事情会悄悄爬到我们身上。

我们都看见过这样的症状。项目慢慢地、不可改变地完全失去控制。大多数软件灾难都是从微不足道的小事情开始的，大多数项目的拖延都是一天一天发生的。系统一个特性一个特性地偏离其规范，一个又一个的补丁被打到某段代码上，直到最初的代码一点没有留下。常常是小事情的累积破坏了士气和团队。

提示 6**Remember the Big Picture**

记住大图景

我们没有做过这个——真的，但有人说，如果你抓一只青蛙放进沸水里，它会一下子跳出来。但是，如果你把青蛙放进冷水里，然后慢慢加热，青蛙不会注意到温度的缓慢变化，会呆在锅里，直到被煮熟。

¹ 这样做时，记住海军少将 Grace Hopper 博士的话，你也许会觉得舒服一点：“请求原谅比获取许可更容易。”

注意，青蛙的问题与第 2 节讨论的破窗户问题不同。在破窗户理论中，人们失去与熵战斗的意愿，是因为他们觉察到没有人会在意。而青蛙只是没有注意到变化。

不要像青蛙一样。留心大图景。要持续不断地观察周围发生的事情，而不只是你自己在做的事情。

相关内容：

- 软件的熵，4 页
- 靠巧合编程，172 页
- 重构，184 页
- 需求之坑，202 页
- 注重实效的团队，224 页

挑战

- 在评阅本书的草稿时，John Lakos 提出这样一个问题：士兵渐进地欺骗村民，但他们所催生的变化对村民完全有利。但是，渐进地欺骗青蛙，你是在加害于它。当你设法催生变化时，你能否确定你是在做石头汤还是青蛙汤？决策是主观的还是客观的？

4 足够好的软件

欲求更好，常把好事变糟。

——李尔王 1.4

有一个（有点）老的笑话，说一家美国公司向一家日本制造商订购 100 000 片集

成电路。规格说明中有次品率：10 000 片中只能有 1 片。几周过后订货到了：一个大盒子，里面装有数千片 IC，还有一个小盒子，里面只装有 10 片 IC。在小盒子上有一个标签，上面写着：“这些是次品”。

要是我们真的能这样控制质量就好了。但现实世界不会让我们制作出十分完美的产品，特别是不会有无错的软件。时间、技术和急躁都在合谋反对我们。

但是，这并不一定就让人气馁。如 Ed Yourdon 发表在 *IEEE Software* 上的一篇文章[You95]所描述的，你可以训练你自己，编写出足够好的软件——对你的用户、对未来的维护者、对你自己内心的安宁来说足够好。你会发现，你变得更多产，而你的用户也会更加高兴。你也许还会发现，因为“孵化期”更短，你的程序实际上更好了。

在继续前进之前，我们需要对我们将要说的话进行限定。短语“足够好”并非意味着不整洁或制作糟糕的代码。所有系统都必须满足其用户的需求，才能取得成功。我们只是在宣扬，应该给用户以机会，让他们参与决定你所制作的东西何时已足够好。

让你的用户参与权衡

通常你是为别人编写软件。你常常需要记得从他们那里获取需求²。但你是否常问他们，他们想要他们的软件有多好？有时候选择并不存在。如果你的工作对象是心脏起搏器、航天飞机、或是将被广泛传播的底层库，需求就会更苛刻，你的选择就更有限。但是，如果你的工作对象是全新的产品，你就会有不同的约束。市场人员有需要信守的承诺，最终用户也许已基于交付时间表制定了各种计划，而你的公司肯定有现金流方面的约束。无视这些用户的需求，一味地给程序增加新特性，或是一次又一次润饰代码，这不是有职业素养的做法。我们不是在提倡慌张：许诺不可能兑现的时间标度（time scale），为赶上最后期限而削减基本的工程内容，这些同样不是有职业素养的做法。

² 从前这被认为是笑话！

你所制作的系统的范围和质量应该作为系统需求的一部分规定下来

提示 7

Make Quality a Requirements Issue

使质量成为需求问题

你常常会处在须要进行权衡的情形中。让人惊奇的是，许多用户宁愿在今天用上有一些“毛边”的软件，也不愿等待一年后的多媒体版本。许多预算吃紧的 IT 部门都会同意这样的说法。今天的了不起的软件常常比明天的完美软件更可取。如果你给用户某样东西，让他们及早使用，他们的反馈常常会把你引向更好的最终解决方案（参见曳光弹，48 页）。

知道何时止步

在某些方面，编程就像是绘画。你从空白的画布和某些基本原材料开始，通过知识、艺术和技艺的结合去确定用前者做些什么。你勾画出全景，绘制背景，然后填入各种细节。你不时后退一步，用批判的眼光观察你的作品。常常，你会扔掉画布，重新再来。

但艺术家们会告诉你，如果你不懂得应何时止步，所有的辛苦劳作就会遭到毁坏。如果你一层又一层、细节复细节地叠加，绘画就会迷失在绘制之中。

不要因为过度修饰和过于求精而毁损完好的程序。继续前进，让你的代码凭着自己的质量站立一会儿。它也许不完美，但不用担心：它不可能完美（在第 6 章，171 页，我们将讨论在不完美的世界上开发代码的哲学）。

相关内容：

- 曳光弹，48 页
- 需求之坑，202 页
- 注重实效的团队，224 页
- 极大的期待，255 页

挑战

- 考察你使用的软件工具和操作系统的制造商。你能否发现证据，表明这些公司安于发布他们知道不完美的软件吗？作为用户，你是会（1）等着他们消除所有 bug，（2）拥有复杂的软件，并接受某些 bug，还是会（3）选择缺陷较少的更简单的软件？
- 考虑模块化对软件交付的影响。与以模块化方式设计的系统相比，整体式（monolithic）软件要达到所需质量，花费的时间更多还是更少？你能找到一个商业案例吗？

5 你的知识资产

知识上的投资总能得到最好的回报。

——本杰明·富兰克林

噢，好样的老富兰克林——从不会想不出精练的说教。为什么，如果我们能够早睡早起，我们就是了不起的程序员——对吗？早起的鸟儿有虫吃，但早起的虫子呢？

然而在这种情况下，Ben 确实命中了要害。你的知识和经验是你最重要的职业财富。

遗憾的是，它们是有时效的资产（expiring asset）³。随着新技术、语言及环境的出现，你的知识会变得过时。不断变化的市场驱动力也许会使你的经验变得陈旧或无关紧要。考虑到“网年”飞逝的速度，这样的事情可能会非常快地发生。

随着你的知识的价值降低，对你的公司或客户来说，你的价值也在降低。我们要阻止这样的事情，决不让它发生。

³ 有时效的资产是价值随时间流逝而逐渐减少的资产，例如香蕉库存和球票。

你的知识资产

我们喜欢把程序员所知道的关于计算技术和他们所工作的应用领域的全部事实、以及他们的所有经验视为他们的知识资产（Knowledge Portfolios）。管理知识资产与管理金融资产非常相似：

1. 严肃的投资者定期投资——作为习惯
2. 多元化是长期成功的关键
3. 聪明的投资者在保守的投资和高风险、高回报的投资之间平衡他们的资产。
4. 投资者设法低买高卖，以获取最大回报。
5. 应周期性地重新评估和平衡资产。

要在职业生涯中获得成功，你必须运用同样的指导方针管理你的知识资产。

经营你的资产

- **定期投资。**就像金融投资一样，你必须定期为你的知识资产投资。即使投资量很小，习惯自身也和总量一样重要。在下一节中将列出一些示范目标。
- **多元化。**你知道的不同的事情越多，你就越有价值。作为底线，你需要知道你目前所用的特定技术的各种特性。但不要就此止步。计算技术的面貌变化很快——今天的热门技术明天就可能变得近乎无用（或至少是不再抢手）。你掌握的技术越多，你就越能更好地进行调整，赶上变化。
- **管理风险。**从高风险、可能有高回报，到低风险、低回报，技术存在于这样一条谱带上。把你所有的金钱都投入可能突然崩盘的高风险股票并不是一个好主意；你也不应太保守，错过可能的机会。不要把你所有的技术鸡蛋放在一个篮子里。
- **低买高卖。**在新兴的技术流行之前学习它可能就和找到被低估的股票一样困难，

但所得到的就和那样的股票带来的收益一样。在 Java 刚出现时学习它可能有风险，但对于现在已步入该领域的顶尖行列的早期采用者，这样做得到了非常大的回报。

- **重新评估和平衡。**这是一个非常动荡的行业。你上个月开始研究的热门技术现在也许已像石头一样冰冷。也许你需要重温你有一阵子没有使用的数据库技术。又或许，如果你之前试用过另一种语言，你就会更有可能获得那个新职位……

在所有这些指导方针中，最重要的也是最简单的：

提示 8

Invest Regularly in Your Knowledge Portfolio
定期为你的知识资产投资

目标

关于何时以及增加什么到你的知识资产中，现在你已经拥有了一些指导方针，那么什么是获得智力资本、从而为你的资产提供资金的最佳方式呢？这里有一些建议。

- **每年至少学习一种新语言。**不同语言以不同方式解决相同的问题。通过学习若干不同的方法，可以帮助你拓宽你的思维，并避免墨守成规。此外，现在学习许多语言已容易了许多，感谢可从网上自由获取的软件财富（参见 267 页）。
- **每季度阅读一本技术书籍。**书店里摆满了许多书籍，讨论与你当前的项目有关的有趣话题。一旦你养成习惯，就一个月读一本书。在你掌握了你正在使用的技术之后，扩宽范围，阅读一些与你的项目无关的书籍。
- **也要阅读非技术书籍。**记住计算机是由人——你在设法满足其需要的人——使用的，这十分重要。不要忘了等式中人这一边。

- **上课。**在本地的学院或大学、或是将要来临的下一次会展上寻找有趣的课程
- **参加本地用户组织。**不要只是去听讲，而要主动参与。与世隔绝对你的职业生涯来说可能是致命的；打听一下你们公司以外的人都在做什么
- **试验不同的环境。**如果你只在 Windows 上工作，就在家玩一玩 Unix（可自由获取的 Linux 正好）。如果你只用过 makefile 和编辑器，就试一试 IDE，反之亦然
- **跟上潮流。**订阅商务杂志和其他期刊（参见 262 页的推荐刊物）。选择所涵盖的技术与你当前的项目不同的刊物。
- **上网。**想要了解某种新语言或其他技术的各种特性？要了解其他人的相关经验，了解他们使用的特定行话，等等，新闻组是一种很好的方式。上网冲浪，查找论文、商业站点，以及其他任何你可以找到的信息来源

持续投入十分重要。一旦你熟悉了某种新语言或新技术，继续前进、学习另一种。

是否在某个项目中使用这些技术，或者是否把它们放入你的简历，这并不重要。学习的过程将扩展你的思维，使你向着新的可能性和新的做事方式拓展。思想的“异花授粉”（cross-pollination）十分重要；设法把你学到的东西应用到你当前的项目中。即使你的项目没有使用该技术，你或许也能借鉴一些想法。例如，熟悉了面向对象，你就会用不同的方式编写纯 C 程序。

学习的机会

于是你狼吞虎咽地阅读，在你的领域，你站在了所有突破性进展的前沿（这不是容易的事情）。有人向你请教一个问题，答案是什么？你连最起码的想法都没有。你坦白地承认了这一点。

不要就此止步，把找到答案视为对你个人的挑战。去请教古鲁（如果在你们的办公

室里没有，你应该能在 Internet 上找到；参见下一页上的方框）。上网搜索 去图书馆⁴。

如果你自己找不到答案，就去找能找到答案的人。不要把问题搁在那里。与他人交谈可以帮助你建立人际网络，而因为在这个过程中找到了其他不相关问题的解决方案，你也许还会让自己大吃一惊。旧有的资产也在不断增长……

所有阅读和研究都需要时间，而时间已经很短缺。所以你需要预先规划。让自己在空闲的片刻时间里总有东西可读。花在等医生上的时间是抓紧阅读的好机会——但一定要带上你自己的杂志，否则，你也许会发现自己在翻阅 1973 年的一篇卷角的关于巴布亚新几内亚的文章。

批判的思考

最后一个要点是，批判地思考你读到的和听到的。你需要确保你的资产中的知识是准确的，并且没有受到供应商或媒体炒作的影响。警惕声称他们的信条提供了惟一答案的狂热者——那或许适用，或许不适用于你和你的项目。

不要低估商业主义的力量。Web 搜索引擎把某个页面列在最前面，并不意味着那就是最佳选择；内容供应商可以付钱让自己排在前面。书店在显著位置展示某一本书，也并不意味着那就是一本好书，甚至也不说明那是一本受欢迎的书；它们可能是付了钱才放在那里的。

提示 9

Critically Analyze What You Read and Hear

批判地分析你读到的和听到的

遗憾的是，几乎再没有简单的答案了。但拥有大量知识资产，并把批判的分析应用于你将要阅读的技术出版物的洪流，你将能够理解复杂的答案。

⁴ 在网络时代，许多人似乎已经忘记了满是研究资料和工作人员的真实图书馆。

与古鲁打交道的礼节与教养

随着 Internet 在全球普及，古鲁们突然变得像你的 Enter 键一样贴近。那么，你怎样才能找到一个古鲁，怎样才能找一个古鲁和你交谈呢？

我们找到了一些简单的诀窍。

- 确切地知道你想要问什么，并尽量明确具体。
- 小心而得体地组织你的问题。记住你是在请求帮助；不要显得好像是在要求对方回答。
- 组织好问题之后，停下来，再找找答案。选出一些关键字，搜索 Web。查找适当的 FAQ（常见问题的解答列表）。
- 决定你是想公开提问还是私下提问。Usenet 新闻组是与专家会面的美妙场所，在那里可以讨论几乎任何问题，但有些人对这些新闻组的公共性质有顾虑。你总是可以用另外的方法：直接发电子邮件给古鲁。不管怎样，要使用有意义的主题（“需要帮助!!!”无益于事）。
- 坐回椅子上，耐心等候。人们很忙，也许需要几天才能得到明确的答案。

最后，请一定要感谢任何回应你的人。如果你看到有人提出你能够解答的问题，尽你的一份力，参与解答。

挑战

- 这周就开始学习一种新语言。总在用 C++ 编程？试试 Smalltalk[URL 13]或 Squeak[URL 14]。在用 Java？试试 Eiffel[URL 10]或 TOM[URL 15]。关于其他自由编译器和环境的来源，参见 267 页。
- 开始阅读一本新书（但要先读完这一本！）。如果你在进行非常详细的实现和编码，就阅读关于设计和架构的书。如果你在进行高级设计，就阅读关于编码技术的书。
- 出去和与你的当前项目无关的人、或是其他公司的人谈谈技术。在你们公司的自助

餐厅里结识其他人，或是在本地用户组织聚会时寻找兴趣相投的人

6 交流！

我相信，被打量比被忽略要好。

——Mae West, *Belle of the Nineties*, 1934

也许我们可以从 West 女士那里学到一点什么。问题不只是你有什么，还要看你怎样包装它。除非你能够与他人交流，否则就算你拥有最好的主意、最漂亮的代码、或是最注重实效的想法，最终也会毫无结果。没有有效的交流，一个好想法就只是一个无人关心的孤儿。

作为开发者，我们必须在许多层面上进行交流。我们把许多小时花在开会、倾听和交谈上。我们与最终用户一起工作，设法了解他们的需要。我们编写代码，与机器交流我们的意图；把我们的想法变成文档，留给以后的开发者。我们撰写提案和备忘录，用以申请资源并证明其正当性、报告我们的状态、以及提出各种新方法。我们每天在团队中工作，宣扬我们的主意、修正现有的做法、并提出新的做法。我们的时间有很大一部分都花在交流上，所以我们需要把它做好。

我们汇总了我们觉得有用的一些想法。

知道你想要说什么

在工作中使用的更为正式的交流方式中，最困难的部分也许是确切地弄清楚你想要说什么。小说家在开始写作之前，会详细地构思情节，而撰写技术文档的人却常常乐于坐到键盘前，键入“1. 介绍……”，并开始敲入接下来在他们的头脑里冒出来的任何东西。

规划你想要说的东西。写出大纲。然后问你自己：“这是否讲清了我要说的所有内容？”提炼它，直到确实如此为止。

这个方法不只适用于撰写文档。当你面临重要会议、或是要与重要客户通电话时，简略记下你想要交流的想法，并准备好几种把它们讲清楚的策略。

了解你的听众

只有当你是在传达信息时，你才是在交流。为此，你需要了解你的听众的需要、兴趣、能力。我们都曾出席过这样的会议：一个做开发的滑稽人物在发表长篇独白，讲述某种神秘技术的各种优点，把市场部副总裁弄得目光呆滞。这不是交流，而只是空谈，让人厌烦的（*annoying*⁵）空谈。

要在脑海里形成一幅明确的关于你的听众的画面。下一页的图 1.1 中显示的 WISDOM 离合诗（*acrostic*）可能会对你有帮助。

假设你想提议开发一个基于 Web 的系统，用于让你们的最终用户提交 bug 报告，取决于听众的不同，你可以用不同的方式介绍这个系统。如果可以不用在电话上等候，每天 24 小时提交 bug 报告，最终用户将会很高兴。你们的市场部门可以利用这一事实促销，支持部门的经理会因为两个原因而高兴：所需员工更少，问题报告得以自动化。最后，开发者会因为能获得基于 Web 的客户 - 服务器技术和新数据库引擎方面的经验而感到享受。通过针对不同的人进行适当的修正，你将让他们都为你的项目感到兴奋。

选择时机

这是星期五的下午六点，审计人员进驻已有一周，你的老板最小的孩子在医院里，外面下着滂沱大雨，这时开车回家肯定是一场噩梦。这大概不是向她提出 PC 内存升级的好时候。

为了了解你的听众需要听到什么，你需要弄清楚他们的“轻重缓急”是什么。找到一个刚刚因为丢失源码而遭到老板批评的经理，向她介绍你关于源码仓库的构想，你将会拥有一个更容易接纳的倾听者。要让你所说的适得其时，在内容上切实相关。有时候，只要简单地问一句“现在我们可以谈谈……吗？”就可以了。

⁵ *annoy* 这个单词源自古法语 *enui*，后者的意思也是“to bore”。

图 1.1 WISDOM 离合诗——了解听众

What do you want them to learn?	你想让他们学到什么?
What is their interest in what you've got to say?	他们对你讲的什么感兴趣?
How sophisticated are they?	他们有多富有经验?
How much detail do they want?	他们想要多少细节?
Whom do you want to own the information?	你想要让谁拥有这些信息?
How can you motivate them to listen to you?	你如何促使他们听你说话?

选择风格

调整你的交流风格，让其适应你的听众。有人要的是正式的“事实”简报。另一些人喜欢在进入正题之前高谈阔论一番。如果是书面文档，则有人喜欢一大摞报告，而另一些人却喜欢简单的备忘录或电子邮件。如果有疑问，就询问对方。

但是，要记住，你也是交流事务的一方。如果有人说，他们需要你用一段话进行描述，而你觉得不用若干页纸就无法做到，如实告诉他们。记住，这样的反馈也是交流的一种形式。

让文档美观

你的主意很重要。它们应该以美观的方式传递给你的听众。

太多程序员（和他们的经理）在制作书面文档时只关心内容。我们认为这是一个错误。任何一个厨师都会告诉你，你可以在厨房里忙碌几个小时，最后却会因为饭菜糟糕的外观而毁掉你的努力。

在今天，已经没有任何借口制作出外观糟糕的打印文档。现代的字处理器（以及像 LaTeX 和 troff 这样的排版系统）能够生成非常好的输出。你只需要学习一些基本的命令。如果你的字处理器支持样式表，就加以利用（你的公司也许已经定义了你可以使用

的样式表) 学习如何设置页眉和页脚。查看你的软件包中包含的样本文档, 以对样式和版式有所了解。检查拼写, 先自动, 再手工。毕竟, 有一些拼写错误是检查器找不出来的 (After awl, their are spelling miss streaks that the chequer can knot ketch)。

让听众参与

我们常常发现, 与制作文档的过程相比, 我们制作出的文档最后并没有那么重要。如果可能, 让你的读者参与文档的早期草稿的制作。获取他们的反馈, 并汲取他们的智慧。你将建立良好的工作关系, 并很可能在此过程中制作出更好的文档。

做倾听者

如果你想要大家听你说话, 你必须使用一种方法: 听他们说话。即使你掌握着全部信息, 即使那是一个正式会议, 你站在 20 个衣着正式的人面前——如果你不听他们说话, 他们也不会听你说话。

鼓励大家通过提问来交谈, 或是让他们总结你告诉他们的东西。把会议变成对话, 你将能更有效地阐明你的观点。谁知道呢, 你也许还能学到点什么。

回复他人

如果你向别人提问, 他们不做出回应, 你会觉得他们不礼貌。但当别人给你发送电子邮件或备忘录, 请你提供信息, 或是采取某种行动时, 你是否经常忘记回复? 在匆忙的日常生活中, 很容易忘记事情。你应该总是对电子邮件和语音邮件做出回应, 即使内容只是“我稍后回复你。”随时通知别人, 会让他们更容易原谅你偶然的疏忽, 并让他们觉得你没有忘记他们。

提示 10

It's Both What You Say and the Way You Say It
你说什么和你怎么说同样重要

除非你生活在真空中, 你才不需要能交流。交流越有效, 你就越有影响力。

电子邮件交流

我们所说的关于书面交流的所有东西都同样适用于电子邮件。现在的电子邮件已经发展成为公司内部和公司之间进行交流的主要手段。它被用于讨论合约、调解争端，以及用作法庭证据。但因为某种原因，许多从不会发出低劣的书面文档的人却乐于往全世界乱扔外观糟糕的电子邮件。

我们关于电子邮件的提示很简单：

- 在你按下 **SEND** 之前进行校对。
- 检查拼写。
- 让格式保持简单。有人使用均衡字体（proportional font）阅读电子邮件，所以你辛苦制作的 ASCII 艺术图形在他们看来将像是母鸡的脚印一样乱七八糟。
- 只在你知道对方能够阅读 rich-text 或 HTML 格式的邮件的情况下使用这些格式。纯文本是通用的。
- 设法让引文减至最少。没有人喜欢收到一封回邮，其中有 100 行是他原来的电子邮件，只在最后新添了三个字：“我同意”。
- 如果你引用别人的电子邮件，一定要注明出处。并在正文中进行引用（而不是当做附件）。
- 不要用言语攻击别人（flame），除非你想让别人也攻击你，并老是纠缠你。
- 在发送之前检查你的收件人名单。最近《华尔街日报》上有一篇文章报道说，有一个雇员通过部门的电子邮件散布对老板的不满，却没有意识到老板也在收件人名单里。
- 将你的电子邮件——你收到的重要文件和你发送的邮件——加以组织并存档。

如 Microsoft 和 Netscape 的好些雇员在 1999 年司法部调查期间所发现的，e-mail 是永久性的。要设法像对待任何书面备忘录或报告一样小心对待 e-mail。

总结

- 知道你想要说什么
- 了解你的听众
- 选择时机
- 选择风格
- 让文档美观
- 让听众参与
- 做倾听者
- 回复他人

相关内容：

- 原型与便笺，53 页
- 注重实效的团队，224 页

挑战

- 有几本好书讨论了开发团队内部的交流[Bro95, McC95, DL99]。下决心在接下来的 18 个月里读完所有这三本书。此外，*Dinosaur Brains*[Ber96]这本书讨论了我们所有人都会带到工作环境中的“情绪包袱”
- 在你下一次进行展示、或是撰写备忘录支持某种立场时，先试着按第 20 页的 **WISDOM** 离合诗做一遍。看这样是否有助于你了解怎样定位你的讲话。如果合适，事后与你的听众谈一谈，看你对他们的需要的估计有多准确

第 2 章

注重实效的途径 A Pragmatic Approach

有些提示和诀窍可应用于软件开发的所有层面，有些想法几乎是公理，有些过程实际上普遍适用。但是，人们几乎没有为这些途径建立这样的文档，你很可能会发现，它们作为零散的段落写在关于设计、项目管理或编码的讨论中。

在这一章里，我们将要把这些想法和过程集中在一起。头两节，“重复的危害”与“正交性”，密切相关。前者提醒你，不要在系统各处对知识进行重复，后者提醒你，不要把任何一项知识分散在多个系统组件中。

随着变化的步伐加快，我们越来越难以让应用跟上变化。在“可撤销性”中，我们将考察有助于使你的项目与其不断变化的环境绝缘的一些技术。

接下来的两节也是相关的。在“曳光弹”中，我们将讨论一种开发方式，能让你同时搜集需求、测试设计、并实现代码。这听起来太好，不可能是真的？的确如此：曳光弹开发并非总是可以应用。“原型与便笺”将告诉你，在曳光弹开发不适用的情况下，怎样使用原型来测试架构、算法、接口以及各种想法。

随着计算机科学慢慢成熟，设计者正在制作越来越高级的语言，尽管能够接受“让它这样”（**make it so**）指令的编译器还没有发明出来，在“领域语言”中我们给出了一些适度的建议，你可以自行加以实施。

最后，我们都是在一个时间和资源有限的世界上工作。如果你善于估计出事情需要多长时间完成，你就能更好地在两者都很匮乏的情况下生存下去（并让你的老板更高兴）。我们将在“估算”中涵盖这一主题。

在开发过程中牢记这些基本原则，你就将能编写更快、更好、更强健的代码。你甚至可以让这看起来很容易。

7 重复的危害

给予计算机两项自相矛盾的知识，是 James T. Kirk 舰长（出自 Star Trek，“星际迷航”——译注）喜欢用来使四处劫掠的人工智能生命失效的方法。遗憾的是，同样的原则也能有效地使你的代码失效。

作为程序员，我们收集、组织、维护和利用知识。我们在规范中记载知识，在运行的代码中使其活跃起来并将其用于提供测试过程中所需的检查。

遗憾的是，知识并不稳定。它变化——常常很快。你对需求的理解可能会随着与客户的会谈而发生变化。政府改变规章制度，有些商业逻辑过时了。测试也许表明所选择的算法无法工作。所有这些不稳定都意味着我们要把很大一部分时间花在维护上，重新组织和表达我们的系统中的知识。

大多数人都以为维护是在应用发布时开始的，维护就意味着修正 bug 和增强特性。我们认为这些人错了。程序员须持续不断地维护。我们的理解逐日变化。当我们设计或编码时，出现了新的需求。环境或许变了。不管原因是什么，维护都不是时有时无的活动，而是整个开发过程中的例行事务。

当我们进行维护时，我们必须找到并改变事物的表示——那些嵌在应用中的知识胶囊。问题是，在我们开发的规范、过程和程序中很容易重复表述知识，而当我们这样做时，我们是在向维护的噩梦发出邀请——在应用发布之前就会开始的噩梦。

我们觉得，可靠地开发软件，并让我们的开发更易于理解和维护的惟一途径，是遵循我们称之为 *DRY* 的原则：

系统中的每一项知识都必须具有单一、无歧义、权威的表示。

我们为何称其为 *DRY*？

提示 11

DRY – Don't Repeat Yourself

不要重复你自己

与此不同的做法是在两个或更多地方表达同一事物。如果你改变其中一处，你必须记得改变其他各处。或者，就像那些异形计算机，你的程序将因为自相矛盾而被迫屈服。这不是你是否能记住的问题，而是你何时忘记的问题。

你会发现 *DRY* 原则在全书中一再出现，并且常常出现在与编码无关的语境中。我们觉得，这是注重实效的程序员工具箱里最重要的工具之一。

在这一节我们将概述重复的问题，并提出对此加以处理的一般策略。

重复是怎样发生的

我们所见到的大多数重复都可归入下列范畴：

- 强加的重复 (**imposed duplication**)。开发者觉得他们无可选择——环境似乎要求重复。
- 无意的重复 (**inadvertent duplication**)。开发者没有意识到他们在重复信息。

- **无耐性的重复 (impatient duplication)**。开发者偷懒，他们重复，因为那样似乎更容易
- **开发者之间的重复 (interdeveloper duplication)**。同一团队（或不同团队）的几个人重复了同样的信息。

让我们更详细地看一看这四个以“i”开头的重复

强加的重复

有时，重复似乎是强加给我们的。项目标准可能要求建立含有重复信息的文档，或是重复代码中的信息的文档。多个目标平台各自需要自己的编程语言、库以及开发环境，这会使我们重复共有的定义和过程。编程语言自身要求某些重复信息的结构。我们都在我们觉得无力避免重复的情形下工作过。然而也有一些方法，可用于把一项知识存放在一处，以遵守 *DRY* 原则，同时也让我们的生活更容易一点。这里有一些这样的技术：

信息的多种表示。在编码一级，我们常常需要以不同的形式表示同一信息。我们也许在编写客户-服务器应用，在客户和服务端使用了不同的语言，并且需要在两端都表示某种共有的结构。我们或许需要一个类，其属性是某个数据库表的 *schema*（模型、方案）的镜像。你也许在撰写一本书，其中包括的程序片段，也正是你要编译并测试的程序。

发挥一点聪明才智，你通常能够消除重复的需要。答案常常是编写简单的过滤器或代码生成器。可以在每次构建（*build*）软件时，使用简单的代码生成器，根据公共的元数据表示构建多种语言下的结构（示例参见图 3.4，106 页）。可以根据在线数据库 *schema*、或是最初用于构建 *schema* 的元数据，自动生成类定义。本书中摘录的代码，由预处理器在我们每次对文本进行格式化时插入。诀窍是让该过程成为主动的，这不能是一次性转换，否则我们会退回到重复数据的情况。

代码中的文档。程序员被教导说，要给代码加上注释：好代码有许多注释。遗憾的是，没有人教给他们，代码为什么需要注释：糟糕的代码才需要许多注释。

DRY 法则告诉我们，要把低级的知识放在代码中，它属于那里；把注释保留给其他的高级说明。否则，我们就是在重复知识，而每一次改变都意味着既要改变代码，也要改变注释。注释将不可避免地变得过时，而不可信任的注释比完全没有注释更糟（关于注释的更多信息，参见全都是写，248 页）。

文档与代码。你撰写文档，然后编写代码，有些东西变了，你修订文档、更新代码。文档和代码都含有同一知识的表示。而我们都知，在最紧张的时候——最后期限在逼近，重要的客户在喊叫——我们往往会推迟文档的更新。

Dave 曾经参与过一个国际电报交换机项目的开发，很容易理解，客户要求提供详尽的测试规范，并要求软件在每次交付时都通过所有测试。为了确保测试准确地反映规范，开发团队用程序方式、根据文档本身生成这些测试。当客户修订他们的规范时，测试套件会自动改变。有一次团队向客户证明了，该过程很健全，生成验收测试在典型情况下只需要几秒钟。

语言问题。许多语言会在源码中强加可观的重复。如果语言使模块的接口与其实现分离，就常常会出现这样的情况。C 与 C++ 有头文件，在其中重复了被导出变量、函数和（C++ 的）类的名称和类型信息。Object Pascal 甚至会在同一文件里重复这些信息。如果你使用远地过程调用或 CORBA[URL 29]，你将会在接口规范与实现它的代码之间重复接口信息。

没有什么简单的技术可用于克服语言的这些需求。尽管有些开发环境通过自动生成头文件、隐藏了对头文件的需要，而 Object Pascal 允许你缩写重复的函数声明，你通常仍受制于给予你的东西。至少对于大多数与语言有关的问题，与实现不一致的头文件将会产生某种形式的编译或链接错误。你仍会弄错事情，但至少，你将在很早的

时候就得到通知。

再思考一下头文件和实现文件中的注释——绝对没有理由在这两种文件之间重复函数或类头注释（header comment）。应该用头文件记载接口问题，用实现文件记载代码的使用者无须了解的实际细节。

无意的重复

有时，重复来自设计中的错误。

让我们看一个来自配送行业的例子。假定我们的分析揭示，一辆卡车有车型、牌照号、司机及其他一些属性。与此类似，发运路线的属性包括路线、卡车和司机。基于这一理解，我们编写了一些类

但如果 Sally 打电话请病假，我们必须改换司机，事情又会怎样呢？Truck 和 DeliverRoute 都包含有司机。我们改变哪一个？显然这样的重复很糟糕。根据底层的商业模型对其进行规范化（normalize）——卡车的底层属性集真的应包含司机？路线呢？又或许我们需要第三种对象，把司机、卡车及路线结合在一起。不管最终的解决方案是什么，我们都应避免这种不规范的数据。

当我们拥有多个互相依赖的数据元素时，会出现一种不那么显而易见的不规范数据。让我们看一个表示线段的类：

```
class Line {  
public:  
    Point start;  
    Point end;  
    double length;  
};
```

第一眼看上去，这个类似乎是合理的。线段显然有起点和终点，并总是有长度（即使长度为零）。但这里有重复。长度是由起点和终点决定的；改变其中一个，长度就会

变化 最好是让长度成为计算字段：

```
class Line {
public:
    Point start;
    Point end;
    double length() { return start.distanceTo(end); }
};
```

在以后的开发过程中，你可以因为性能原因而选择违反 *DRY* 原则。这经常会发生在你需要缓存数据，以避免重复昂贵的操作时。其诀窍是使影响局部化。对 *DRY* 原则的违反没有暴露给外界：只有类中的方法需要注意“保持行为良好”。

```
class Line {
private:
    bool changed;
    double length;
    Point start;
    Point end;

public:
    void setStart(Point p) { start = p; changed = true; }
    void setEnd(Point p) { end = p; changed = true; }

    Point getStart(void) { return start; }
    Point getEnd(void) { return end; }

    double getLength() {
        if (changed) {
            length = start.distanceTo(end);
            changed = false;
        }
        return length;
    }
};
```

这个例子还说明了像 Java 和 C++ 这样的面向对象语言的一个重要问题。在可能的情况下，应该总是用访问器（accessor）函数读写对象的属性⁶。这将使未来增加功能（比如缓存）变得更容易。

⁶ 访问器函数的使用与 Meyer 的 *Uniform Access* 原则[Mey97b] 紧密相关，该原则规定：“模块提供的所有服务都应能通过统一的表示法使用，该表示法不能泄漏它们是通过存储、还是通过计算实现的。”

无耐性的重复

每个项目都有时间压力——这是能够驱使我们中间最优秀的人走捷径的力量。需要与你写过的一个例程相似的例程？你会受到诱惑，去拷贝原来的代码，并做出一些改动。需要一个表示最大点数的值？如果我改动头文件，整个项目就得重新构建。也许我应该在这里使用直接的数字 (literal number)，这里，还有这里，需要一个与 Java runtime 中的某个类相似的类？源码在那里（你有使用许可），那么为什么不拷贝它、并做出你所需的改动呢？

如果你觉得受到诱惑，想一想古老的格言：“欲速则不达”。你现在也许可以节省几秒钟，但以后却可能损失几小时。想一想围绕着 Y2K 惨败的种种问题。其中许多问题是由开发者的懒惰造成的：他们没有参数化日期字段的尺寸，或是实现集中的日期服务库。

无耐性的重复是一种容易检测和处理的重复杂形式，但那需要你接受训练，并愿意为避免以后的痛苦而预先花一些时间。

开发者之间的重复

另一方面，或许是最难检测和处理的重复杂发生在项目的不同开发者之间。整个功能集都可能在无意中被重复，而这些重复可能几年里都不会被发现，从而导致各种维护问题。我们亲耳听说过，美国某个州在对政府的计算机系统进行 Y2K 问题检查时，审计者发现有超出 10 000 个程序，每一个都有自己的社会保障号验证代码。

在高层，可以通过清晰的设计、强有力的技术项目领导（参见 288 页“注重实效的团队”一节中的内容）、以及在设计中进行得到了充分理解的责任划分，对这个问题加以处理。但是，在模块层，问题更加隐蔽。不能划入某个明显的责任区域的常用功能和数据可能会被实现许多次。

我们觉得，处理这个问题的最佳方式是鼓励开发者相互进行主动的交流，设置论

坛，用以讨论常见问题（在过去的一些项目中，我们设置了私有的 Usenet 新闻组，用于让开发者交换意见，进行提问。这提供了一种不受打扰的交流方式——甚至跨越多个站点——同时又保留了所有言论的永久历史）。让某个团队成员担任项目资料管理员，其工作是促进知识的交流。在源码树中指定一个中央区域，用于存放实用例程和脚本。一定要阅读他人的源码与文档，不管是非正式的，还是进行代码复查。你不是在窥探——你是在向他们学习。而且要记住，访问是互惠的——不要因为别人钻研（乱钻？）你的代码而苦恼。

提示 12

Make It Easy to Reuse
让复用变得容易

你所要做的是营造一种环境，在其中要找到并复用已有的东西，比自己编写更容易。如果不容易，大家就不会去复用。而如果不进行复用，你们就会有重复知识的风险。

相关内容：

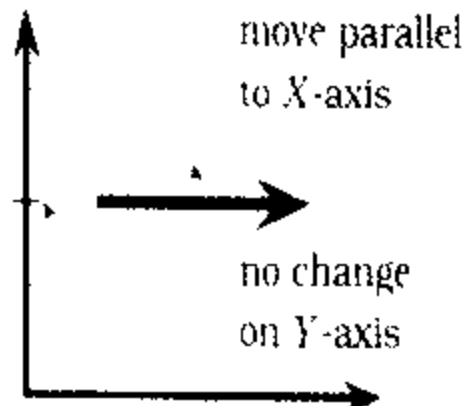
- 正交性，34 页
- 文本操纵，99 页
- 代码生成器，102 页
- 重构，184 页
- 注重实效的团队，224 页
- 无处不在的自动化，230 页
- 全都是写，248 页

8 正交性

如果你想要制作易于设计、构建、测试及扩展的系统，正交性是一个十分关键的概念，但是，正交性的概念很少被直接讲授，而常常是你学习的各种其他方法和技术的隐含特性。这是一个错误。一旦你学会了直接应用正交性原则，你将发现，你制作的系统的质量立刻就得到了提高。

什么是正交性

“正交性”是从几何学中借来的术语。如果两条直线相交成直角，它们就是正交的，比如图中的坐标轴。用向量术语说，这两条直线互不依赖。沿着某一条直线移动，你投影到另一条直线上的位置不变。



在计算技术中，该术语用于表示某种不相依赖性或是解耦性。如果两个或更多事物中的一个发生变化，不会影响其他事物，这些事物就是正交的。在设计良好的系统中，数据库代码与用户界面是正交的：你可以改动界面，而不影响数据库；更换数据库，而不用改动界面。

在我们考察正交系统的好处之前，让我们先看一看非正交系统。

非正交系统

你正乘坐直升机游览科罗拉多大峡谷，驾驶员——他显然犯了一个错误，在吃鱼，他的午餐——突然呻吟起来，晕了过去。幸运的是，他把你留在了离地面 100 英尺的地方。你推断，升降杆⁷控制总升力，所以轻轻将其压低可以让直升机平缓降向地面。

⁷ 直升机有四种基本控制器：转向杆是你握在右手中的手柄。移动它，直升机就会向着相应的方向移动。你的左手握着升降杆，抬起它，你就能增加所有桨叶的倾斜度，产生升力。在升降杆的末端是油门。最后，你还有两个脚踏板，用于改变尾翼的推力大小，从而让直升机转向。

然而，当你这样做时，却发现生活并非那么简单。直升机的鼻子向下，开始向左盘旋下降。突然间你发现，你驾驶的这个系统，所有的控制输入都有次级效应。压低左手的操作杆，你需要补偿性地向后移动右手柄，并踩右踏板。但这些改变中的每一项都会再次影响所有其他的控制。突然间，你在用一个让人难以置信的复杂系统玩杂耍，其中每一项改变都会影响所有其他的输入。你的工作负担异常巨大：你的手脚在不停地移动，试图平衡所有交互影响的力量。

直升机的各个控制器断然不是正交的。

正交的好处

如直升机的例子所阐明的，非正交系统的改变与控制更复杂是其固有的性质。当任何系统的各组件互相高度依赖时，就不再有局部修正（local fix）这样的事情。

提示 13

Eliminate Effects Between Unrelated Things

消除无关事物之间的影响

我们想要设计自足（self-contained）的组件：独立，具有单一、良好定义的目的（Yourdon 和 Constantine 称之为内聚（cohesion）[YC86]）。如果组件是相互隔离的，你就知道你能够改变其中之一，而不用担心其余组件。只要你不改变组件的外部接口，你就可以放心：你不会造成波及整个系统的问题。

如果你编写正交的系统，你得到两个主要好处：提高生产率与降低风险。

提高生产率

- 改动得以局部化，所以开发时间和测试时间得以降低。与编写单个的大块代码相比，编写多个相对较小的、自足的组件更为容易。你可以设计、编写简单的组件，

对其进行单元测试，然后把它们忘掉——当你增加新代码时，无须不断改动已有的代码

- 正交的途径还能够促进复用。如果组件具有明确而具体的、良好定义的责任，就可以用其最初的实现者未曾想象过的方式，把它们与新组件组合在一起
- 如果你对正交的组件进行组合，生产率会有相当微妙的提高。假定某个组件做 M 件事情，而另一个组件做 N 件事情。如果它们是正交的，而你把它们组合在一起，结果就能做 $M \times N$ 件事情。但是，如果这两个组件是非正交的，它们就会重叠，结果能做的事情就更少。通过组合正交的组件，你的每一份努力都能得到更多的功能

降低风险

正交的途径能降低任何开发中固有的风险。

- 有问题的代码区域被隔离开来。如果某个模块有毛病，它不大可能把病症扩散到系统的其余部分。要把它切掉，换成健康的新模块也更容易。
- 所得系统更健壮。对特定区域做出小的改动与修正，你所导致的任何问题都将局限在该区域中。
- 正交系统很可能能得到更好的测试，因为设计测试、并针对其组件运行测试更容易。
- 你不会与特定的供应商、产品、或是平台紧绑在一起，因为与这些第三方组件的接口将被隔离在全部开发的较小部分中。

让我们看一看在工作中应用正交原则的几种方式。

项目团队

你是否注意到，有些项目团队很有效率，每个人都知道要做什么，并全力做出贡献

献，而另一些团队的成员却老是在争吵，而且好像无法避免互相妨碍？

这常常是一个正交性问题。如果团队的组织有许多重叠，各个成员就会对责任感到困惑。每一次改动都需要整个团队开一次会，因为他们中的任何一个人都可能受到影响。

怎样把团队划分为责任得到了良好定义的小组，并使重叠降至最低呢？没有简单的答案，这部分地取决于项目本身，以及你对可能变动的区域的分析。这还取决于你可以得到的人员。我们的偏好是从使基础设施与应用分离开始。每个主要的基础设施组件（数据库、通信接口、中间件层，等等）有自己的子团队。如果应用功能的划分显而易见，那就照此划分。然后我们考察我们现有的（或计划有的）人员，并对分组进行相应的调整。

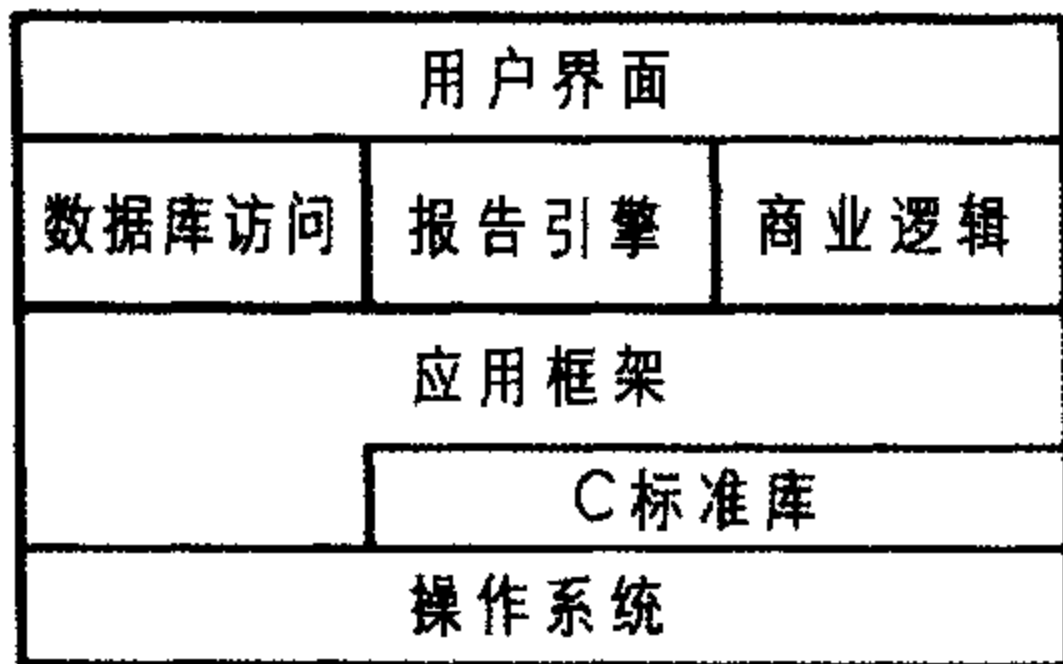
你可以对项目团队的正交性进行非正式的衡量。只要看一看，在讨论每个所需改动时需要涉及多少人。人数越多，团队的正交性就越差。显然，正交的团队效率也更高（尽管如此，我们也鼓励子团队不断地相互交流）。

设计

大多数开发者都熟知需要设计正交的系统，尽管他们可能会使用像模块化、基于组件、或是分层这样的术语描述该过程。系统应该由一组相互协作的模块组成，每个模块都实现不依赖于其他模块的功能。有时，这些组件被组织为多个层次，每层提供一级抽象。这种分层的途径是设计正交系统的强大方式。因为每层都只使用在其下面的层次提供的抽象，在改动底层实现、而又不影响其他代码方面，你拥有极大的灵活性。分层也降低了模块间依赖关系失控的风险。你将常常看到像下一页的图 2.1 这样的图表示的层次关系。

对于正交设计，有一种简单的测试方法。一旦设计好组件，问问你自己：如果我显著地改变某个特定功能背后的需求，有多少模块会受影响？在正交系统中，答案应

图 2.1 典型的层次图



该是“一个”⁸ 移动 GUI 面板上的按钮，不应该要求改动数据库 schema 增加语境敏感的帮助，也不应该改动记账子系统

让我们考虑一个用于监视和控制供暖设备的复杂系统。原来的需求要求提供图形用户界面，但后来需求被改为要增加语音应答系统，用按键电话控制设备。在正交地设计的系统中，你只需要改变那些与用户界面有关联的模块，让它们对此加以处理：控制设备的底层逻辑保持不变。事实上，如果你仔细设计你的系统结构，你应该能够用同一个底层代码库支持这两种界面。157 页的“它只是视图”将讨论怎样使用模型 - 视图 - 控制器（MVC）范型编写解耦的代码，该范型在这里的情况下也能很好地工作。

⁸ 在现实中，这有点天真。除非你非常幸运，否则大多数实际的需求变动都会影响系统中的多个功能。但是，如果你根据功能分析变动，每个功能的变动在理论上仍然应该只影响一个模块。

还要问问你自己，你的设计在多大程度上解除了与现实世界中的变化的耦合？你在把电话号码当作顾客标识符吗？如果电话公司重新分配了区号，会怎么样？不要依赖你无法控制的事物属性。

工具箱与库

在你引入第三方工具箱和库时，要注意保持系统的正交性。要明智地选择技术。

我们曾经参加过一个项目，在其中需要一段 Java 代码，既运行在本地的服务器机器上，又运行在远地的客户机器上。要把类按这样的方式分布，可以选用 RMI 或 CORBA。如果用 RMI 实现类的远地访问，对类中的远地方法的每一次调用都可能会抛出异常；这意味着，一个幼稚的实现可能会要求我们，无论何时使用远地类，都要对异常进行处理。在这里，使用 RMI 显然不是正交的：调用远地类的代码应该不用知道这些类的位置，另一种方法——使用 CORBA——就没有施加这样的限制：我们可以编写不知道我们类的位置的代码。

在引入某个工具箱时（甚或是来自你们团队其他成员的库），问问你自己，它是否会迫使你对代码进行不必要的改动。如果对象持久模型（object persistence scheme）是透明的，那么它就是正交的。如果它要求你以一种特殊的方式创建或访问对象，那么它就不是正交的。让这样的细节与代码隔离具有额外的好处：它使得你在以后更容易更换供应商。

Enterprise Java Beans (EJB) 系统是正交性的一个有趣例子。在大多数面向事务的系统中，应用代码必须描述每个事务的开始与结束。在 EJB 中，该信息是作为元数据，在任何代码之外，以声明的方式表示的。同一应用代码不用修改，就可以运行在不同的 EJB 事务环境中。这很可能是将来许多环境的模型。

正交性的另一个有趣的变体是面向方面编程（Aspect-Oriented Programming, AOP），这是 Xerox Parc 的一个研究项目（[KLM+97]与[URL 49]）。AOP 让你在一个地方表达本来会分散在源码各处的某种行为。例如，日志消息通常是在源码各处、通

过显式地调用某个日志函数生成的。通过 AOP，你把日志功能正交地实现到要进行日志记录的代码中。使用 AOP 的 Java 版本，你可以通过编写 *aspect*，在进入类 Fred 的任何方法时写日志消息：

```
aspect Trace {
  advise * Fred.*(..) {
    static before {
      Log.writef("-> Entering " + thisJoinPoint.methodName);
    }
  }
}
```

如果你把这个方面编织（*weave*）进你的代码，就会生成追踪消息。否则，你就不会看到任何消息。不管怎样，你原来的源码都没有变化。

编码

每次你编写代码，都有降低应用正交性的风险。除非你不仅时刻监视你正在做的事情，也时刻监视应用的更大语境，否则，你就有可能无意中重复其他模块的功能，或是两次表示已有的知识。

你可以将若干技术用于维持正交性：

- **让你的代码保持解耦。**编写“羞怯”的代码——也就是不会没有必要地向其他模块暴露任何事情、也不依赖其他模块的实现的模块。试一试我们将在 183 页的“解耦与得墨忒耳法则”中讨论的得墨忒耳法则（*Law of Demeter*）[LH89]。如果你需要改变对象的状态，让这个对象替你去做。这样，你的代码就会保持与其他代码的实现的隔离，并增加你保持正交的机会。
- **避免使用全局数据。**每当你的代码引用全局数据时，它都把自己与共享该数据的其他组件绑在了一起。即使你只想对全局数据进行读取，也可能会带来麻烦（例如，如果你突然需要把代码改为多线程的）。一般而言，如果你把所需的任何语境

(context) 显式地传入模块, 你的代码就会更易于理解和维护。在面向对象应用中, 语境常常作为参数传给对象的构造器。换句话说, 你可以创建含有语境的结构, 并传递指向这些结构的引用。

《设计模式》[GHJV95]一书中的 Singleton (单体) 模式是确保特定类的对象只有一个实例的一种途径。许多人把这些 singleton 对象用作某种全局变量 (特别是在除此而外不支持全局概念的语言中, 比如 Java)。使用 singleton 要小心——它们可能造成不必要的关联。

- **避免编写相似的函数。**你常常会遇到看起来全都很像的一组函数——它们也许在开始和结束处共享公共的代码, 中间的算法却各有不同。重复的代码是结构问题的一种症状。要了解更好的实现, 参见《设计模式》一书中的 Strategy (策略) 模式。

养成不断地批判对待自己的代码的习惯。寻找任何重新进行组织、以改善其结构和正交性的机会。这个过程叫做重构 (refactoring), 它非常重要, 所以我们专门写了一节加以讨论 (见“重构”, 184 页)。

测试

正交地设计和实现的系统也更易于测试, 因为系统的各组件间的交互是形式化的和有限的, 更多的系统测试可以在单个的模块级进行。这是好消息, 因为与集成测试 (integration testing) 相比, 模块级 (或单元) 测试要更容易规定和进行得多。事实上, 我们建议让每个模块都拥有自己的、内建在代码中的单元测试, 并让这些测试作为常规构建过程的一部分自动运行 (参见“易于测试的代码”, 189 页)。

构建单元测试本身是对正交性的一项有趣测试。要构建和链接某个单元测试, 都需要什么? 只是为了编译或链接某个测试, 你是否就必须把系统其余的很大一部分拽进来? 如果是这样, 你已经发现了一个没有很好地解除与系统其余部分耦合的模块。

修正 bug 也是评估整个系统的正交性的好时候。当你遇到问题时, 评估修正的局部化程度。

你是否只改动了一个模块, 或者改动分散在整个系统的各个地方? 当你做出改动

时，它修正了所有问题，还是又神秘地出现了其他问题？这是开始运用自动化的好机会。如果你使用了源码控制系统（在阅读了 86 页的“源码控制”之后，你会使用的），当你在测试之后、把代码签回（`check the code back`）时，标记所做的 bug 修正。随后你可以运行月报，分析每个 bug 修正所影响的源文件数目的变化趋势。

文档

也许会让人惊讶，正交性也适用于文档。其坐标轴是内容和表现形式。对于真正正交的文档，你应该能显著地改变外观，而不用改变内容。现代的字处理器提供了样式表和宏，能够对你有帮助（参见“全都是写”，248 页）。

认同正交性

正交性与 27 页介绍的 *DRY* 原则紧密相关。运用 *DRY* 原则，你是在寻求使系统中的重复降至最小；运用正交性原则，你可降低系统的各组件间的相互依赖。这样说也许有点笨拙，但如果你紧密结合 *DRY* 原则、运用正交性原则，你将会发现你开发的系统会变得更加灵活、更易于理解、并且更易于调试、测试和维护。

如果你参加了一个项目，大家都在不顾一切地做出改动，而每一处改动似乎都会造成别的东西出错，回想一下直升机的噩梦，项目很可能没有进行正交的设计和编码，是重构的时候了。

另外，如果你是直升机驾驶员，不要吃鱼……

相关内容：

- 重复的危害，26 页
- 源码控制，86 页
- 按合约设计，109 页
- 解耦与得墨忒耳法则，138 页
- 元程序设计，144 页
- 它只是视图，157 页

- 重构, 184 页
- 易于测试的代码, 189 页
- 邪恶的向导, 198 页
- 注重实效的团队, 224 页
- 全都是写, 248 页

挑战

- 考虑常在 Windows 系统上见到的面向 GUI 的大型工具和在 shell 提示下使用的短小、但却可以组合的命令行实用工具。哪一种更为正交, 为什么? 如果正好按其设计用途加以应用, 哪一种更易于使用? 哪一种更易于与其他工具组合, 以满足新的要求?
- C++ 支持多重继承, 而 Java 允许类实现多重接口。使用这些设施对正交性有何影响? 使用多重继承与使用多重接口的影响是否有不同? 使用委托 (delegation) 与使用继承之间是否有不同?

练习

1. 你在编写一个叫做 Split 的类, 其用途是把输入行拆分为字段。下面的两个 Java 类的型构 (signature) 中, 哪一个是为更正交的设计? (解答在 279 页)

```
class Split1 {
    public Split1(InputStreamReader rdr) { ...
    public void readNextLine() throws IOException { ...
    public int numFields() { ...
    public String getField(int fieldNo) { ...
}
class Split2 {
    public Split2(String line) { ...
    public int numFields() { ...
    public String getField(int fieldNo) { ...
}
```

2. 非模态对话框或模态对话框, 哪一个能带来更为正交的设计? (解答在 279 页)
3. 过程语言与对象技术的情况又如何? 哪一种能产生更为正交的系统? (解答在 280 页)

9 可撤消性

如果某个想法是你惟一的想法，再没有什么比这更危险的事情了

——Emil-Auguste Chartier, *Propos sur la religion*, 1938

工程师们喜欢问题有简单、单一的解决方案。与论述法国大革命的无数起因的一篇模糊、热烈的文章相比，允许你怀着极大的自信宣称 $x=2$ 的数学测验要让人觉得舒服得多。管理人员往往与工程师趣味相投：单一、容易的答案正好可以放在电子表格和项目计划中。

现实世界能够合作就好了！遗憾的是，今天 x 是 2，明天也许就需要是 5，下周则是 3。没有什么永远不变——而如果你严重依赖某一事实，你几乎可以确定它将会变化。

要实现某种东西，总有不只一种方式，而且通常有不只一家供应商可以提供第三方产品。如果你参与的项目被短视的、认为只有一种实现方式的观念所牵绊，你也许就会遇到让人不悦的意外之事。许多项目团队会被迫在未来展现之时睁开眼睛：

“但你说过我们要使用 XYZ 数据库！我们的项目已经完成了 85% 的编码工作，我们现在不能改变了！”程序员抗议道。“对不起，但我们公司决定进行标准化，改用 PDQ 数据库——所有项目。这超出了我的职权范围，我们必须重新编码。周末所有人都要加班，直到另行通知为止。”

变动不一定会这么严苛，甚至也不会这么迫在眉睫。但随着时间的流逝，随着你的项目取得进展，你也许会发现自己陷在无法立足的处境里。随着每一项关键决策的做出，项目团队受到越来越小的目标的约束——现实的更窄小的版本，选择的余地越来越小。

在许多关键决策做出之后，目标会变得如此之小，以至于如果它动一下，或是风改变方向，或是东京的蝴蝶扇动翅膀，你都会错过目标⁹。而且你可能会偏出很远。

⁹ 取一个非线性的（混沌的）系统，将一点小变动应用于其输入之一，你可能会得到巨大的、常常是无法预测的结果。一个老套的例子：一只蝴蝶在东京扇动翅膀，就可能引起链式反应，并最终造成得克萨斯的一场龙卷风。这听起来是否像你知道的某个项目？

问题在于，关键决策不容易撤销

一旦你决定使用这家供应商的数据库、那种架构模式、或是特定的部署模型（例如，客户 - 服务器 vs. 单机），除非付出极大的代价，否则你就将受制于一个无法撤销的动作进程（course of action）

可撤销性

我们让本书的许多话题相互配合，以制作灵活、有适应能力的软件——通过遵循它们的建议——特别是 *DRY* 原则（26 页）、解耦（138 页）以及元数据的使用（144 页）——我们不必做出许多关键的、不可逆转的决策。这是一件好事情，因为我们并非总能在一开始就做出最好的决策。我们采用了某种技术，却发现我们雇不到足够的具有必需技能的人。我们刚刚选定某个第三方供应商，他们就被竞争者收购了。与我们开发软件的速度相比，需求、用户以及硬件变得更快

假定在项目初期，你决定使用供应商 A 提供的关系数据库。过了很久，在性能测试过程中，你发现数据库简直太慢了，而供应商 B 提供的对象数据库更快。对于大多数传统项目，你不会有什么运气。大多数时候，对第三方产品的调用都缠绕在代码各处。但如果你真的已经把数据库的概念抽象出来——抽象到数据库只是把持久（persistence）作为服务提供出来的程度——你就会拥有“中流换马（change horses in midstream）”的灵活性。

与此类似，假定项目最初采用的是客户 - 服务器模型，但随即，在开发的后期，市场部门认为服务器对于某些客户过于昂贵，他们想要单机版。对你来说，那会有多困难？因为这只是一个部署问题，所以不应该要很多天。如果所需时间更长，那么你就没有考虑过可撤销性。另外一个方向甚至更有趣。如果需要以客户 - 服务器或 n 层方式部署你正在开发的单机产品，事情又会怎样？那也不应该很困难

错误在于假定决策是浇铸在石头上的——同时还在于没有为可能出现的意外事件做准备

要把决策视为是写在沙滩上的，而不要把它们刻在石头上。大浪随时可能到来，把它们抹去。

提示 14

There Are No Final Decisions

不存在最终决策

灵活的架构

有许多人会设法保持代码的灵活性，而你还需要考虑维持架构、部署及供应商集成等领域的灵活性。

像 CORBA 这样的技术可以帮助把项目的某些部分与开发语言或平台的变化隔离开来。Java 在该平台上的性能不能满足要求？重新用 C++ 编写客户代码，其他没有什么需要改变。用 C++ 编写的规则引擎不够灵活？换到 Smalltalk 版本。采用 CORBA 架构，你只须改动替换的组件；其他组件应该不会受影响。

你正在开发 UNIX 软件？哪一种？你是否处理了所有可移植性问题？你正在为某个特定版本的 Windows 做开发？哪一种——3.1、95、98、NT、CE，或是 2000？支持其他版本有多难？如果你让决策保持软和与柔韧，事情就完全不困难。如果在代码中有着糟糕的封装、高度耦合以及硬编码的逻辑或参数，事情也许就是不可能的。

不确定市场部门想怎样部署系统？预先考虑这个问题，你可以支持单机、客户-服务器，或 n 层模型——只需要改变配置文件。我们就写过一些这么做的程序。

通常，你可以把第三方产品隐藏在定义良好的抽象接口后面。事实上，在我们做过的任何项目中，我们都总能够这么做。但假定你无法那么彻底地隔离它，如果你必须大量地把某些语句分散在整个代码中，该怎么办？把该需求放入元数据，并且使用某种自动机制——比如 Aspect（参见 39 页）或 Perl——把必需的语句插入代码自身。

中 无论你使用的是何种机制，让它可撤销。如果某样东西是自动添加的，它也可以被自动去掉。

没有人知道未来会怎样，尤其是我们！所以要让你的代码学会“摇滚”：可以“摇”就“摇”，必须“滚”就“滚”。

相关内容：

- 解耦与得墨忒耳法则，138 页
- 元程序设计，144 页
- 它只是视图，157 页

挑战

- 让我们通过“薛定谔的猫”学一点量子力学。假定在一个封闭的盒子里有一只猫，还有一个放射性粒子。这个粒子正好有 50% 的机会裂变成两个粒子。如果发生了裂变，猫就会被杀死；如果没有，猫就不会有事。那么，猫是死是活？根据薛定谔的理论，正确的答案是“都是”。每当有两种可能结果的亚核反应发生时，宇宙就会被克隆。在其中一个宇宙中，事件发生；在另一个宇宙中，事件不发生。猫在一个宇宙中是活的，在另一个宇宙中是死的。只有当你打开盒子，你才知道你在哪一个宇宙里。

怪不得为未来编码很困难。

但想一想，代码沿着与装满薛定谔的猫的盒子一样的路线演化：每一项决策都会导致不同版本的未来。你的代码能支持多少种可能的未来？哪一种未来更有可能发生？到时支持它们有多困难？

你敢打开盒子吗？

10 曳光弹

预备、开火、瞄准……

在黑暗中用机枪射击有两种方式¹⁰。你可以找出目标的确切位置（射程、仰角及方位）。你可以确定环境状况（温度、湿度、气压、风，等等）。你可以确定你使用的弹药筒和子弹的精确规格，以及它们与你使用的机枪的交互作用。然后你可以用计算表或射击计算机计算枪管的确切方向及仰角。如果每一样东西都严格按照规定的方式工作，你的计算表正确无误，而且环境没有发生变化，你的子弹应该能落在距目标不远的地方。

或者，你可以使用曳光弹。

曳光弹与常规弹药交错着装在弹药带上。发射时，曳光弹中的磷点燃，在枪与它们击中的地方之间留下一条烟火般的踪迹。如果曳光弹击中目标，那么常规子弹也会击中目标。

并不让人惊奇的是，曳光弹比费力计算更可取。反馈是即时的，而且因为它们工作在与真正的弹药相同的环境中，外部影响得以降至最低。

这个类比也许有点暴力，但它适用于新的项目，特别是当你构建从未构建过的东西时。与枪手一样，你也设法在黑暗中击中目标。因为你的用户从未见过这样的系统，他们的需求可能会含糊不清。因为你在使用不熟悉的算法、技术、语言或库，你面对着大量未知的事物，同时，因为完成项目需要时间，在很大程度上你能够确知，你的工作环境将在你完成之前发生变化。

经典的做法是把系统定死，制作大量文档，逐一列出每项需求、确定所有未知因素、并限定环境。根据死的计算射击。预先进行一次大量计算，然后射击并企望击中。

¹⁰ 如果要咬文嚼字，在黑暗中用机枪射击有许多方式，包括闭上双眼胡乱扫射。但这只是一个类比，我们可以随意一点。

目标。

然而，注重实效的程序员往往更喜欢使用曳光弹。

在黑暗中发光的代码

曳光弹行之有效，是因为它们与真正的子弹在相同的环境、相同的约束下工作。它们快速飞向目标，所以枪手可以得到即时的反馈。同时，从实践的角度看，这样的解决方案也更便宜。

为了在代码中获得同样的效果，我们要找到某种东西，让我们能快速、直观和可重复地从需求出发，满足最终系统的某个方面要求。

提示 15

Use Tracer Bullets to Find the Target

用曳光弹找到目标

有一次，我们接受了一个复杂的客户-服务器数据库营销项目。其部分需求是要能够指定并执行临时查询。服务器是一系列专用的关系数据库。用 Object Pascal 编写的客户 GUI 使用一组 C 库提供给服务器的接口。在转换为优化的 SQL 之前，用户的查询以类似 Lisp 的表示方式存储在服务器上；转换直到执行前才进行。有许多未知因素和许多不同的环境，没有人清楚地知道 GUI 应该怎样工作。

这是使用曳光代码的好机会。我们开发了前端框架、用于表示查询的库以及用于把所有存储的查询转换为具体数据库的查询的结构。随后我们把它们集中在一起，并检查它们是否能工作。使用最初构建的系统，我们所能做的只是提交一个查询，列出某个表中的所有行，但它证明了 UI 能够与库交谈，库能够对查询进行序列化和解序列化，而服务器能够根据结果生成 SQL。在接下来的几个月里，我们逐渐充实这个基本结构，通过并行地扩大曳光代码的各个组件增加新的功能。当 UI 增加了新的查询类型时，库随之成长，而我们也使 SQL 生成变得更为成熟。

曳光代码并非用过就扔的代码：你编写它，是为了保留它。它含有任何一段产品代码都拥有的完整的错误检查、结构、文档、以及自查。它只不过功能不全而已。但是，一旦你在系统的各组件间实现了端到端（end-to-end）的连接，你就可以检查你离目标还有多远，并在必要的情况下进行调整。一旦你完全瞄准，增加功能将是一件容易的事情。

曳光开发与项目永不会结束的理念是一致的：总有改动需要完成，总有功能需要增加。这是一个渐进的过程。

另一种传统做法是一种繁重的工程方法：把代码划分为模块，在真空中对模块进行编码。把模块组合成子配件（subassembly），再对子配件进行组合，直到有一天你拥有完整的应用为止。直到那时，才能把应用作为一个整体呈现给用户，并进行测试。

曳光代码方法有许多优点：

- **用户能够及早看到能工作的东西。**如果你成功地就你在做的事情与用户进行了交流（参见“极大的期望”，255页），用户就会知道他们看到的是还未完成的东西。他们不会因为缺少功能而失望；他们将因为看到了系统的某种可见的进展而欣喜陶醉。他们还会随着项目的进展做出贡献，增加他们的“买入”。同样是这些用户，他们很可能也会告诉你，每一轮“射击”距离目标有多接近。
- **开发者构建了一个他们能在其中工作的结构。**最令人畏缩的纸是什么也没有写的白纸。如果你已经找出应用的所有端到端的交互，并把它们体现在代码里，你的团队就无须再无中生有。这让每个人都变得更有生产力，同时又促进了一致性。
- **你有了一个集成平台。**随着系统端到端地连接起来，你拥有了一个环境，一旦新的代码段通过了单元测试，你就可以将其加入该环境中。你将每天进行集成（常常是一天进行多次），而不是尝试进行大爆炸式的集成。每一个新改动的影响都更为显而易见，而交互也更为有限，于是调试和测试将变得更快、更准确。

- **你有了可用于演示的东西。**项目出资人与高级官员往往会在最不方便的时候来看演示。有了曳光代码，你总有东西可以拿给他们看。
- **你将更能够感觉到工作进展。**在曳光代码开发中，开发者一个一个地处理用例（use case），做完一个，再做下一个。评测性能、并向用户演示你的进展，变得容易了许多。因为每一项个别的开发都更小，你也避免了创建这样的整体式代码块：一周又一周，其完成度一直是 95%。

曳光弹并非总能击中目标

曳光弹告诉你击中的是什么。那不一定总是目标。于是你调整准星，直到完全击中目标为止。这正是要点所在。

曳光代码也是如此。你在不能 100%确定该去往何处的情形下使用这项技术。如果最初的几次尝试错过了目标——用户说：“那不是我的意思”，你需要的数据在你需要它时不可用，或是性能好像有问题——你不应感到惊奇。找出怎样改变已有的东西、让其更接近目标的办法，并且为你使用了一种简约的开发方法而感到高兴。小段代码的惯性也小——要改变它更容易、更迅速。你能够搜集关于你的应用的反馈，而且与其他任何方法相比，你能够花费较少代价、更为迅速地生成新的、更为准确的版本。同时，因为每个主要的应用组件都已表现在你的曳光代码中，用户可以确信，他们所看到的东西具有现实基础，不仅仅是纸上的规范。

曳光代码 vs. 原型制作

你也许会想，这种曳光代码的概念就是原型制作，只不过有一个更富“进攻性”的名字。它们有区别。使用原型，你是要探究最终系统的某些具体的方面。使用真正的原型，在对概念进行了试验之后，你会把你捆扎在一起的无论什么东西扔掉，并根据你学到的经验教训重新适当地进行编码。

例如，假定你在制作一个应用，其用途是帮助运货人确定怎样把不规则的箱子装入集装箱。

除了考虑其他一些问题，你还需要设计直观的用户界面，而你用于确定最优装箱方式的算法非常复杂

你可以在 GUI 工具中为最终用户制作一个用户界面原型。你的代码只能让界面响应用户操作。一旦用户对界面布局表示同意，你可以把它扔掉，用目标语言重新对其进行编码，并在其后加上商业逻辑。与此类似，你可以为实际进行装箱的算法制作原型。你可以用像 Perl 这样的宽松的高级语言编写功能测试，并用更接近机器的某种语言编写低级的性能测试。无论如何，一旦你做出决策，你都会重新开始在其最终环境中为算法编写代码，与现实世界接合。这就是原型制作，它非常有用。

曳光代码方法处理的是不同的问题。你需要知道应用怎样结合成一个整体。你想要向用户演示，实际的交互是怎样工作的，同时你还想要给出一个架构骨架，开发者可以在其上增加代码。在这样的情况下，你可以构造一段曳光代码，其中含有一个极其简单的集装箱装箱算法实现（也许是像“先来先服务”这样的算法）和一个简单，但却能工作的用户界面。一旦你把应用中的所有组件都组合在一起，你就拥有了一个可以向你的用户和开发者演示的框架。接下来的时间里，你给这个框架增加新功能，完成预留了接口的例程。但框架仍保持完整，而你也不知道，系统将会继续按照你第一次的曳光代码完成时的方式工作。

其间的区别很重要，足以让我们再重复一次。原型制作生成用过就扔的代码。曳光代码虽然简约，但却是完整的，并且构成了最终系统的骨架的一部分。你可以把原型制作视为在第一发曳光弹发射之前进行的侦察和情报搜集工作。

相关内容：

- 足够好的软件，9 页
- 原型与便笺，53 页
- 规范陷阱，217 页
- 极大的期望，255 页

11 原型与便笺

许多不同的行业都使用原型试验具体的想法：与完全的制作相比，制作原型要便宜得多。例如，轿车制造商可以制造某种新车设计的许多不同的原型，每一种的设计目的都是要测试轿车的某个具体的方面——空气动力学、样式、结构特征，等等。也许会制造一个粘土模型，用于风洞测试，也许会为工艺部门制造一个轻木和胶带模型，等等。有些轿车公司更进一步，在计算机上进行大量的建模工作，从而进一步降低了开销。以这样的方式，可以试验危险或不确定的元件，而不用实际进行真实的制造。

我们以同样的方式构建软件原型，并且原因也一样——为了分析和揭示风险，并以大大降低的代价，为修正提供机会。与轿车制造商一样，我们可以把原型用于测试项目的一个或多个具体方面。

我们往往以为原型要以代码为基础，但它们并不总是非如此不可。与轿车制造商一样，我们可以用不同的材料构建原型。要为像工作流和应用逻辑这样的动态事物制作原型，便笺（post-it note）就非常好。用户界面的原型则可以是白板上的图形、或是用绘图程序或界面构建器绘制的无功能的模型。

原型的设计目的就是回答一些问题，所以与投入使用的产品应用相比，它们的开发要便宜得多、快捷得多。其代码可以忽略不重要的细节——在此刻对你不重要，但对后来的用户可能非常重要。例如，如果你在制作 GUI 原型，你不会因不正确的结果或数据而遭到指责，而另一方面，如果你只是在研究计算或性能方面的问题，你也不会因为相当糟糕的 GUI 而遭到指责；甚至也可以完全不要 GUI。

但如果你发现自己处在不能放弃细节的环境中，就需要问自己，是否真的在构建原型。或许曳光弹开发方式更适合这种情况（参见“曳光弹”，48 页）。

应制作原型的事物

你可以选择通过原型来研究什么样的事物呢？任何带有风险的事物。以前没有试过的事物，或是对于最终系统极端关键的事物。任何未被证明的、实验性的、或有疑问的事物。任何让你觉得不舒服的事物。你可以为下列事物制作原型：

- 架构
- 已有系统中的新功能
- 外部数据的结构或内容
- 第三方工具或组件
- 性能问题
- 用户界面设计

原型制作是一种学习经验。其价值并不在于所产生的代码，而在于所学到的经验教训。那才是原型制作的要点所在。

提示 16

Prototype to Learn

为了学习而制作原型

怎样使用原型

在构建原型时，你可以忽略哪些细节？

- **正确性**。你也许可以在适当的地方使用虚设的数据。
- **完整性**。原型也许只能在非常有限的意义上工作，也许只有一项预先选择的输入数据和一个菜单项。
- **健壮性**。错误检查很可能不完整，或是完全没有。如果你偏离预定路径，原型就可能崩溃，并在“烟火般的灿烂显示中焚毁”。这没有关系。
- **风格**。在纸上承认这一点让人痛苦，但原型代码可能没有多少注释或文档。根据使用原型的经验，你也许会撰写出大量文档，但关于原型系统自身的内容相对而言却

非常少

因为原型应该遮盖细节，并聚焦于所考虑系统的某些具体方面，你可以用非常高级的语言实现原型——比项目的其余部分更高级（也许是像 Perl、Python 或 Tcl 这样的语言）。高级的脚本语言能让你推迟考虑许多细节（包括指定数据类型），并且仍然能制作出能工作的（即使不完整或速度慢）代码¹¹。如果你需要制作用户界面的原型，可研究像 Tcl/Tk、Visual Basic、Powerbuilder 或 Delphi 这样的工具。

作为能把低级的部分组合在一起的“胶合剂”，脚本语言工作良好。在 Windows 下，Visual Basic 可以把 COM 控件胶合在一起。更一般地说，你可以使用像 Perl 和 Python 这样的语言，把低级的 C 库绑在一起——无论是手工进行，还是通过工具自动进行，比如可以自由获取的 SWIG[URL 28]。采用这种方法，你可以快速地把现有组件装配进新的配置，从而了解它们的工作情况。

制作架构原型

许多原型被构造出来，是要为在考虑之下的整个系统建模。与曳光弹不同，在原型系统中，单个模块不需要能行使特定的功能。事实上，要制作架构原型，你甚至不一定需要进行编码——你可以用便笺或索引卡片、在白板上制作原型。你寻求的是了解系统怎样结合成为一个整体，并推迟考虑细节。下面是一些你可以在架构原型中寻求解答的具体问题：

- 主要组件的责任是否得到了良好定义？是否适当？
- 主要组件间的协作是否得到了良好定义？
- 耦合是否得以最小化？
- 你能否确定重复的潜在来源？
- 接口定义和各项约束是否可接受？

¹¹ 如果你在探查绝对的（而不是相对的）性能，你需要坚持采用性能与目标语言接近的语言。

- 每个模块在执行过程中是否能访问到其所需的数据？是否能在需要时进行访问？

根据我们制作原型的经验，最后一项往往会产生最让人惊讶和最有价值的结果

怎样“不”使用原型

在你着手制作任何基于代码的原型之前，先确定每个人都理解你正在编写用过就扔的代码。对于不知道那只是原型的人，原型可能会具有欺骗性的吸引力。你必须非常清楚地说明，这些代码是用过就扔的，它们不完整，也不可能完整。

别人很容易被演示原型外表的完整性误导，而如果你没有设定正确的期望值，项目出资人或管理部门可能会坚持要部署原型（或其后裔）。提醒他们，你可以用轻木和胶带制造一辆了不起的新车原型，但你却不会在高峰时间的车流中驾驶它。

如果你觉得在你所在的环境或文化中，原型代码的目的很有可能被误解，你也许最好还是采用曳光弹方法。你最后将得到一个坚实的框架，为将来的开发奠定基础。

适当地使用原型，可以帮助你早期确定和改正潜在的问题点——在此时改正错误既便宜、又容易——从而为你节省大量时间、金钱，并大大减轻你遭受的痛苦和折磨。

相关内容：

- 我的源码让猫给吃了，2 页
- 交流！，18 页
- 曳光弹，48 页
- 极大的期望，255 页

练习

4. 市场部门想要坐下来和你一起讨论一些网页的设计问题。他们想用可点击的图像进行页面导航，但却不能确定该用什么图像模型——也许是轿车、电话或是房子。你

有一些目标网页和内容；他们想要看到一些原型。哦，随便说一下，你只有 15 分钟。你可以使用什么样的工具？（解答在 280 页）

12 领域语言

语言的界限就是一个人的世界的界限。

——维特根斯坦

计算机语言会影响你思考问题的方式，以及你看待交流的方式。每种语言都含有一系列特性——比如静态类型与动态类型、早期绑定与迟后绑定、继承模型（单、多或无）这样的时髦话语——所有这些特性都在提示或遮蔽特定的解决方案。头脑里想着 Lisp 设计的解决方案将会产生与基于 C 风格的思考方式而设计的解决方案不同的结果，反之亦然。与此相反——我们认为这更重要——问题领域的语言也可能会提示出编程方案。

我们总是设法使用应用领域的语汇来编写代码（参见 210 页的需求之坑，我们在那里提出要使用项目词汇表）。在某些情况下，我们可以更进一层，采用领域的语汇、语法、语义——语言——实际进行编程。

当你听取某个提议中的系统的用户说明情况时，他们也许能确切地告诉你，系统应怎样工作：

在一组 X.25 线路上侦听由 ABC 规程 12.3 定义的交易，把它们转译成 XYZ 公司的 43B 格式，在卫星上行链路上重新传输，并存储起来，供将来分析使用。

如果用户有一些这样的做了良好限定的陈述，你可以发明一种为应用领域进行了适当剪裁的小型语言，确切地表达他们的需要：

```
From X25LINE1 (Format=ABC123) {
  Put TELSTAR1 (Format=XY/43B);
  Store DB;
}
```


该语言无须是可执行的。一开始，它可以只是用于捕捉用户需求的一种方式——一种规范。但是，你可能想要更进一步，实际实现该语言。你的规范变成了可执行代码。

在你编写完应用之后，用户给了你一项新需求：不应存储余额为负的交易，而应以原来的格式在 X.25 线路上发送回去：

```
From X25LINE1 (Format=ABC123) {
  if (ABC123.balance < 0) {
    Put X25LINE1 (Format=ABC123);
  }
  else {
    Put TELSTAR1 (Format=XYZ43B);
    Store DB;
  }
}
```

很容易，不是吗？有了适当的支持，你可以用大大接近应用领域的方式进行编程。我们并不是在建议让你的最终用户用这些语言实际编程。相反，你给了自己一个工具，能够让你更靠近他们的领域工作。

提示 17

Program Close to the Problem domain

靠近问题领域编程

无论是用于配置和控制应用程序的简单语言，还是用于指定规则或过程的更为复杂的语言，我们认为，你都应该考虑让你的项目更靠近问题领域。通过在更高的抽象层面上编码，你获得了专心解决领域问题的自由，并且可以忽略琐碎的实现细节。

记住，应用有许多用户。有最终用户，他们了解商业规则和所需输出；也有次级用户：操作人员、配置与测试管理人员、支持与维护程序员，还有将来的开发者。他们都有各自的问题领域，而你可以为他们所有人生成小型环境和语言。

具体领域的错误

如果你是在问题领域中编写程序，你也可以通过用户可以理解的术语进行具体领域的验证，或是报告问题。以上一页我们的交换应用为例，假定用户拼错了格式名：

```
From X25LINE_ (Format-AB123)
```

如果这发生在某种标准的、通用的编程语言中，你可能会收到一条标准的、通用的错误消息：

```
Syntax error: undeclared identifier
```

但使用小型语言，你却能够使用该领域的语汇发出错误消息：

```
"AB123" is not a format. known formats are ABC 23,  
XYZ43B, PQRB, and 42.
```

实现小型语言

在最简单的情况下，小型语言可以采用面向行的、易于解析的格式。在实践中，与其他任何格式相比，我们很可能会更多地使用这样的格式。只要使用 `switch` 语句，或是使用像 `Perl` 这样的脚本语言中的正则表达式，就能够对其进行解析。281 页上练习 5 的解答给出了一种用 `C` 编写的简单实现。

你还可以用更为正式的语法，实现更为复杂的语言。这里的诀窍是首先使用像 `BNF`¹² 这样的表示法定义语法。一旦规定了文法，要将其转换为解析器生成器（`parser generator`）的输入语法通常就非常简单了。`C` 和 `C++` 程序员多年来一直在使用 `yacc`（或其可自由获取的实现，`bison`[URL 27]）在 *Lex and Yacc*[LMB92]一书中详细地讲述了这些程序。`Java` 程序员可以选用 `javaCC`，可在[URL 26]处获取该程序。282

¹² Backus-Naur Form，可用于递归地规定上下文无关（context-free）的文法。任何关于编译器构造或解析的好书都会详细地（无遗漏地）涵盖 BNF。

页上练习 7 的解答给出了一个用 **bison** 编写的解析器。如其所示，一旦你了解了语法，编写简单的小型语言实在没有多少工作要做。

要实现小型语言还有另一种途径：扩展已有的语言。例如，你可以把应用级功能与 Python[URL 9]集成在一起，编写像这样的代码¹³：

```
record = X25LINE1.get(format=ABC123)
if (record.balance < 0):
    X25LINE1.put(record, format=ABC123)
else:
    TELSTAR1.put(record, format=XYZ43B)
    DB.store(record)
```

数据语言与命令语言

可以通过两种不同的方式使用你实现的语言。

数据语言产生某种形式的数据结构给应用使用。这些语言常用于表示配置信息。

例如，**sendmail** 程序在世界各地被用于在 Internet 上转发电子邮件。它具有许多杰出的特性和优点，由一个上千行的配置文件控制，用 **sendmail** 自己的配置语言编写：

```
Mlocal, P=/usr/bin/procmail,
      F=!sDFMAw5 :/ @qSPfIm9,
      S=10/30, R=20/40,
      T=DNS/RFC822/X Unix,
      A=procmail -Y -a $h -d $u
```

显然，可读性不是 **sendmail** 的强项。

多年以来，Microsoft 一直在使用一种可以描述菜单、**widget**（窗口小部件）、对话框及其他 Windows 资源的数据语言。下一页上的图 2.2 摘录了一段典型的资源文件。这比 **sendmail** 的配置文件要易读得多，但其使用方式却完全一样——我们编译它，以生成数据结构。

命令语言更进了一步。在这种情况下，语言被实际执行，所以可以包含语句、控制结构、以及类似的东西（比如 58 页上的脚本）。

¹³ 感谢 Eric Vought 提供这个例子。

图 2.2 Windows .rc 文件

```

MAIN_MENU MENU
{
    POPUP "&File"
    {
        MENUITEM "&New", CM_FILENEW
        MENUITEM "&Open...", CM_FILEOPEN
        MENUITEM "&Save", CM_FILESAVE
    }
}

MY_DIALOG_BOX DIALOG 6, 15, 292, 287
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE |
        WS_CAPTION | WS_SYSMENU
CAPTION "My Dialog Box"
FONT 8, "MS Sans Serif"
{
    DEFPUSHBUTTON "OK", ID_OK, 232, 16, 50, 14
    PUSHBUTTON "Help", ID_HELP, 232, 52, 50, 14
    CONTROL "Edit Text Control", ID_EDIT1,
        "EDIT", WS_BORDER | WS_TABSTOP, 16, 16, 80, 56
    CHECKBOX "Checkbox", ID_CHECKBOX1, 153, 65, 42, 38,
        BS_AUTOCHECKBOX | WS_TABSTOP
}

```

你也可以使用自己的命令语言来使程序易于维护。例如，也许用户要求你把来自某个遗留应用的信息集成进你的新 GUI 开发中。要完成这一任务，常用的方法是“刮屏”（screen scraping）：你的应用连接到主机应用，就好像它是正常的使用人员；发出键击，并“阅读”取回的响应。你可以使用一种小型语言来把这样的交互编写成脚本¹⁴：

```

locate prompt "SSN:"
type "%s" social_security_number
type enter

waitfor keyboardunlock

if text_at(10,14) is "INVALID SSN" return bad_ssn
if text_at(10,14) is "DUPLICATE SSN" return dup_ssn
# etc...

```

当应用确定是时候输入社会保障号时，它调用解释器执行这个脚本，后者随即对事务进行控制。如果解释器是嵌入在应用中的，两者甚至可以直接共享数据（例如，

¹⁴ 事实上，你可以购买支持这样的脚本编写的工具。你还可以研究像 Expect 这样的开放源码包，它们提供了类似的能力[URL 24]。

通过回调机制)

这里你是在维护程序员 (maintenace programmer) 的领域中编程。当主机应用发生变化、字段移往别处时, 程序员只需更新你的高级描述, 而不用钻入 C 代码的各种细节中。

独立语言与嵌入式语言

要发挥作用, 小型语言无须由应用直接使用。许多时候, 我们可以使用规范语言创建各种由程序自身编译、读入或用于其他用途的制品 (包括元数据。参见元程序设计, 144 页)。

例如, 在 100 页我们将描述一个系统, 在其中我们使用 Perl、根据原始的 schema 规范生成大量衍生物。我们发明了一种用于表示数据库 schema 的通用语言, 然后生成我们所需的所有形式——SQL、C、网页、XML, 等等。应用不直接使用规范, 但它依赖于根据规范产生的输出。

把高级命令语言直接嵌入你的应用是一种常见做法, 这样, 它们就会在你的代码运行时执行。这显然是一种强大的能力; 通过改变应用读取的脚本, 你可以改变应用的行为, 却完全不用编译。这可以显著地简化动态的应用领域中的维护工作。

易于开发还是易于维护

我们已经看到若干不同的文法, 范围从简单的面向行的格式到更为复杂的、看起来像真正的语言的文法。既然实现更为复杂的文法需要额外的努力, 你又为何要这样做呢?

权衡要素是可扩展性与维护。尽管解析“真正的”语言所需的代码可能更难编写, 但它却容易被人理解得多, 并且将来用新特性和新功能进行扩展也要容易得多。太简单的语言也许容易解析, 但却可能晦涩难懂——很像是 60 页上的 sendmail 例子。

考虑到大多数应用都会超过预期的使用期限, 你可能最好咬紧牙关, 先就采用更复杂、可读性更好的语言。最初的努力将在降低支持与维护费用方面得到许多倍的回报。

相关内容:

- 元程序设计, 144 页

挑战

- 你目前的项目的某些需求是否能以具体领域的语言表示? 是否有可能编写编译器或转译器, 生成大多数所需代码?
- 如果你决定采用小型语言作为更接近问题领域的编程方式, 你就是接受了, 实现它们需要一些努力。你能否找到一些途径, 通过它们把你为某个项目开发的框架复用于其他项目?

练习

5. 我们想实现一种小型语言, 用于控制一种简单的绘图包 (或许是一种“海龟图形” (turtle-graphics) 系统) 这种语言由单字母命令组成。有些命令后跟单个数字。例如, 下面的输入将会绘制出一个矩形:

```
P 2 # select pen 2
D   # pen down
W 2 # draw west 2cm
N 1 # then north 1
E 2 # then east 2
S 1 # then back south
U   # pen up
```

请实现解析这种语言的代码。它应该被设计成能简单地增加新命令 (解答在 281 页)

6. 设计一种解析时间规范的 BNF 文法, 应能接受下面的所有例子: (解答在 282 页)

```
4pm, 7:38pm, 23:42, 3:16, 3:16am
```

7. 用 yacc、bison 或类似的解析器生成器为练习 6 中的 BNF 文法实现解析器。(解答在 282 页)
8. 用 Perl 实现时间解析器 (提示: 正则表达式可带来好的解析器)。(解答在 283 页)

13 估算

快！通过 56k modem 线发送《战争与和平》需要多少时间？存储一百万个姓名与地址需要多少磁盘空间？1 000 字节的数据块通过路由器需要多少时间？交付你的项目需要多少个月？

在某种程度上，这些都是没有意义的问题——它们都缺少信息。然而它们仍然可以得到回答，只要你习惯于进行估算。同时，在进行估算的过程中，你将会加深对你的程序所处的世界的理解。

通过学习估算，并将此技能发展到你对事物的数量级有直觉的程度，你就能展现出一种魔法般的能力，确定它们的可行性。当有人说“我们将通过 ISDN 线路把备份发给中央站点”时，你将能够直觉地知道那是否实际。当你编码时，你将能够知道哪些子系统需要优化，哪些可以放在一边。

提示 18

Estimate to Avoid Surprises

估算，以避免发生意外

作为奖励，在这一节的末尾我们将透露一个总是正确的答案——无论什么时候有人要你进行估算，你都可以给出答案。

多准确才足够准确

在某种程度上，所有的解答都是估算。只不过有一些要比其他的更准确。所以当有人要你进行估算时，你要问自己的第一个问题就是，你解答问题的语境是什么？他们是需要高度的准确性，还是在考虑棒球场的大小？

- 如果你的奶奶问你何时抵达，她也许只是想知道该给你准备午餐还是晚餐。而一个困在水下、空气就快用光的潜水员很可能对精确到秒的答案更感兴趣。

- π 的值是多少？如果你想知道的是要买多少饰边，才能把一个圆形花坛围起来，那么“3”很可能就足够好了¹⁵。如果你在学校里，那么“22/7”也许就是一个好的近似值。如果你在 NASA（美国国家航空航天管理局），那么也许要 12 个小数位。

关于估算，一件有趣的事情是，你使用的单位会对结果的解读造成影响。如果你说，某事需要 130 个工作日，那么大家会期望它在相当接近的时间里完成。但是，如果你说“哦，大概要六个月”，那么大家知道它会在从现在开始的五到七个月内完成。这两个数字表示相同的时长，但“130 天”却可能暗含了比你的感觉更高的精确程度。我们建议你这样度量时间估算：

时长	报出估算的单位
1-15 天	天
3-8 周	周
8-30 周	月
30+ 周	在给出估算前努力思考一下

于是，在完成了所有必要的工作之后，你确定项目将需要 125 个工作日（25 周），你可以给出“大约六个月”的估算。

同样的概念适用于对任何数量的估算：要选择能反映你想要传达的精确度的单位。

估算来自哪里

所有的估算都以问题的模型为基础。但在我们过深地卷入建模技术之前，我们必须先提及一个基本的估算诀窍，它总能给出好的答案：去问已经做过这件事情的人。在你一头钻进建模之前，仔细在周围找找也曾处在类似情况下的人。

¹⁵ 如果你是一位议员，“3”显然也足够好了。在 1897 年，印第安那州议院第 246 号议案试图规定，自此以后， π 的值应该是 3。该议案在二读时被无限期搁置，一位数学教授指出，他们的权力不能扩展至通过自然法则。

看看他们的问题是怎么解决的。你不大可能找到完全相符的案例，但你会惊奇有多少次，你能够成功地借鉴他人的经验。

理解提问内容

任何估算练习的第一步都是建立对提问内容的理解。除了上面讨论的精确度问题以外，你还需要把握问题域的范围。这常常隐含在问题中，但你需要养成在开始猜想之前先思考范围的习惯。常常，你选择的范围将形成你给出的解答的一部分：“假定没有交通意外，而且车里还有汽油，我会在 20 分钟内赶到那里。”

建立系统的模型

这是估算有趣的部分。根据你对所提问题的理解，建立粗略、就绪的思维模型骨架。如果你是在估算响应时间，你的模型也许要涉及服务器和某种到达流量（arriving traffic）。对于一个项目，模型可以是你的组织在开发过程中所用的步骤、以及系统的实现方式的非常粗略的图景。

建模既可以是创造性的，又可以是长期有用的。在建模的过程中，你常常会发现一些在表面上不明显的底层模式与过程。你甚至可能会想要重新检查原来的问题：“你要求对做 X 所需的时间进行估算。但好像 X 的变种 Y 只需一半时间就能完成，而你只会损失一个特性。”

建模把不精确性引入了估算过程中。这是不可避免的，而且也是有益的。你是在用模型的简单性与精确性做交易。使花在模型上的努力加倍也许只能带来精确性的轻微提高。你的经验将告诉你何时停止提炼。

把模型分解为组件

一旦拥有了模型，你可以把它分解为组件。你须要找出描述这些组件怎样交互的数学规则。有时某个组件会提供一个值，加入到结果中。有些组件有着成倍的影响，

而另一些可能会更为复杂（比如那些模拟某个节点上的到达流量的组件）。

你将会发现，在典型情况下，每个组件都有一些参数，会对它给整个模型带来什么造成影响。在这一阶段，只要确定每个参数就行了。

给每个参数指定值

一旦你分解出各个参数，你就可以逐一给每个参数赋值。在这个步骤中你可能会引入一些错误。诀窍是找出哪些参数对结果的影响最大，并致力于让它们大致正确。在典型情况下，其值被直接加入结果的参数，没有被乘或除的那些参数重要。让线路速度加倍可以让 1 小时内接收的数据量加倍，而增加 5 毫秒的传输延迟不会有显著的效果。

你应该采用一种合理的方式计算这些关键参数。对于排队的例子，你可以测量现有系统的实际事务到达率，或是找一个类似的系统进行测量。与此类似，你可以测量现在服务 1 个请求所花的时间，或是使用这一节描述的技术进行估算。事实上，你常常会发现自己以其他子估算为基础进行估算。这是最大的错误伺机溜进来的地方。

计算答案

只有在最简单的情况下估算才有单一的答案。你也许会高兴地说：“我能在 15 分钟内走完五个街区。”但是，当系统变得更为复杂时，你就会避免做出正面回答。进行多次计算，改变关键参数的值，直到你找出真正主导模型的那些参数。电子表格可以有很大帮助。然后根据这些参数表述你的答案。“如果系统拥有 SCSI 总线和 64MB 内存，响应时间约为四分之三秒；如果内存是 48MB，则响应时间约为一秒。”（注意“四分之三秒”怎样给人以一种与 750 毫秒不同的精确感。）

在计算阶段，你可能会得到看起来很奇怪的答案。不要太快放弃它们。如果你的运算是正确的，那你对问题或模型的理解就很可能是错的。这是非常宝贵的信息。

追踪你的估算能力

我们认为，记录你的估算，从而让你看到自己接近正确答案的程度，这是一个非常好的主意。如果总体估算涉及子估算的计算，那么也要追踪这些子估算。你常常会发现自己的估算得非常好——事实上，一段时间之后，你就会开始期待这样的事情。

如果结果证明估算错了，不要只是耸耸肩走开。找出事情为何与你的猜想不同的原因。也许你选择了与问题的实际情况不符的一些参数。也许你的模型是错的。不管原因是什么，花一点时间揭开所发生的事情。如果你这样做了，你的下一次估算会更好。

估算项目进度

在面对相当大的应用开发的各种复杂问题与反复无常的情况时，普通的估算规则可能会失效。我们发现，为项目确定进度表的惟一途径常常是在相同的项目上获取经验。如果你实行增量开发、重复下面的步骤，这不一定就是一个悖论：

- 检查需求
- 分析风险
- 设计、实现、集成
- 向用户确认

一开始，你对需要多少次迭代、或是需要多少时间，也许只有模糊的概念。有些方法要求你把这个作为初始计划的一部分定下来，但除了最微不足道的项目，这是一个错误。除非你在开发与前一个应用类似的应用，拥有同样的团队和同样的技术，否则，你就只不过是在猜想。

于是你完成了初始功能的编码与测试，并将此标记为第一轮增量开发的结束。基于这样的经验，你可以提炼你原来对迭代次数、以及在每次迭代中可以包含的内容的猜想。提炼会变得一次比一次好，对进度表的信心也将随之增长。

提示 19**Iterate the Schedule with the Code****通过代码对进度表进行迭代**

这也许并不会受到管理部门的欢迎，在典型情况下，他们想要的是单一的、必须遵守的数字——甚至是在项目开始之前。你必须帮助他们了解团队、团队的生产率、还有环境将决定进度。通过使其形式化，并把改进进度表作为每次迭代的一部分，你将给予他们你能给予的最精确的进度估算。

在被要求进行估算时说什么

你说：“我等会儿回答你。”

如果你放慢估算的速度，并花一点时间仔细检查我们在这一节描述的步骤，你几乎总能得到更好的结果。在咖啡机旁给出的估算将（像咖啡一样）回来纠缠你。

相关内容

- 算法速度，177 页

挑战

- 开始写估算日志。追踪每一次估算的精确程度。如果你的错误率大于 50%，设法找出你的估算误入歧途的地方。

练习

9. 有人问你：“1Mbps 的通信线路和在口袋里装了 4GB 磁带。在两台计算机间步行的人，哪一个的带宽更高？”你要对你的答案附加什么约束，以确保你的答复的范围是正确的？（例如，你可以说，访问磁带所花时间忽略不计。）（解答在 283 页）
10. 那么，哪一个带宽更高？（解答在 284 页）

第 3 章

基本工具

The Basic Tools

每个工匠在开始其职业生涯时，都会准备一套品质良好的基本工具。木匠可能需要尺、计量器、几把锯子、几把好刨子、精良的凿子、钻孔器和夹子、锤子还有钳子。这些工具将经过认真挑选、打造得坚固耐用、并用于完成很少与其他工具重合的特定工作，而且，也许最重要的是，刚刚出道的木匠把它们拿在手里会觉得很顺手。

随后学习与适应的过程就开始了。每样工具都有自身的特性和古怪之处，并且需要得到相应的特殊对待。每样工具都需要以独特的方式进行打磨，或者以独特的方式把持。随着时间的过去，每样工具都会因使用而磨损，直到手柄看上去就像是木匠双手的模子，而切割面与握持工具的角度完全吻合。到这时，工具变成了工匠的头脑与所完成的产品之间的通道——它们变成了工匠双手的延伸。木匠将不时增添新的工具，比如饼式切坯机、激光制导斜切锯、楔形模具——全都是奇妙的技术，但你可以肯定的是，当他把原来的某样工具拿在手里，当他听到刨子滑过木料发出的歌声时，那是他最高兴的时候。

工具放大你的才干。你的工具越好，你越是能更好地掌握它们的用法，你的生产力就越高。从一套基本的通用工具开始，随着经验的获得，随着你遇到一些特殊需求，你将会在其中增添新的工具。要与工匠一样，想着定期增添工具。要总是寻找更好的做事方式。如果你遇到某种情况，你觉得现有的工具不能解决问题，记得去寻找可能会有帮助的其他工具或更强大的工具。

让需要驱动你的采购

许多新程序员都会犯下错误，采用单一的强力工具，比如特定的集成开发环境（IDE），而且再也不离开其舒适的界面。这实在是个错误。我们要乐于超越 IDE 所施加的各种限制。要做到这一点，惟一的途径是保持基本工具集的“锋利”与就绪。

在本章我们将讨论怎样为你自己的基本工具箱投资。与关于工具的任何好的讨论一样，我们将从考察你的原材料——你将要制作的东西——开始（在“纯文本的威力”中）。然后我们将从那里转向工作台（workbench），在我们的工作范围也就是计算机。要怎样使用计算机，你才能最大限度地利用你所用的工具？我们将在 shell 游戏中讨论这一问题。现在我们有了工作所需的材料及工作台，我们将转向一样你可能用得最频繁的工具：你的编辑器。在强力编辑中，我们将提出多种让你更有效率的途径。

为了确保不会丢失先前的任何工作成果，我们应该总是使用源码控制系统——即使是像我们的个人地址簿这样的东西！同时，因为 Murphy 先生实在是一个乐观主义者，如果你没有高超的调试技能，你就不可能成为了不起的程序员。

你需要一些“胶合剂”，把大量魔术“粘”在一起。我们将在文本操纵中讨论一些可能的方案，比如 awk、Perl 以及 Python。

就如同木匠有时会制作模具，用以控制复杂工件的打造一样，程序员也可以编写自身能编写代码的代码。我们将在“代码生成器”中讨论这一问题。

花时间学习使用这些工具，有一天你将会惊奇地发现，你的手指在键盘上移动，操纵文本，却不用进行有意识的思考。工具将变成你的双手的延伸。

14 纯文本的威力

作为**注重实效的程序员**，我们的基本材料不是木头，不是铁，而是知识。我们搜集需求，将其变为知识，随后又在我们的设计、实现、测试、以及文档中表达这些知识。而且我们相信，持久地存储知识的最佳格式是纯文本。通过纯文本，我们给予了自己既能以手工方式、也能以程序方式操纵知识的能力——实际上可以随意使用每一样工具。

什么是纯文本

纯文本由可打印字符组成，人可以直接阅读和理解其形式。例如，尽管下面的片段由可打印字符组成，它却是无意义的：

```
Field19=467abe
```

阅读者不知道 **467abe** 的含义是什么。更好的选择是让其变得能让人理解：

```
DrawingType=UMLActivityDrawing
```

纯文本并非意味着文本是无结构的；XML、SGML 和 HTML 都是有良好定义的结构的好例子。通过纯文本，你可以做你通过某种二进制格式所能做的每件事情，其中包括版本管理。

与直接的二进制编码相比，纯文本所处的层面往往更高；前者通常直接源自实现。假定你想要存储叫做 **uses_menus** 的属性，其值既可为 **TRUE**，也可为 **FALSE**。使用纯文本，你可以将其写为：

```
myprop.uses_menus=FALSE
```

把它与 **0010010101110101** 对比一下。

大多数二进制格式的问题在于，理解数据所必需的语境与数据本身是分离的。你人为地使数据与其含义脱离开来。数据也可能加了密；没有应用逻辑对其进行解析，这些数据绝对没有意义。但是，通过纯文本，你可以获得自描述（self-describing）的、不依赖于创建它的应用的数据流。

提示 20**Keep Knowledge in Plain Text**

用纯文本保存知识

缺点

使用纯文本有两个主要缺点：(1) 与压缩的二进制格式相比，存储纯文本所需空间更多。(2) 要解释及处理纯文本文件，计算上的代价可能更昂贵。

取决于你的应用，这两种情况或其中之一可能让人无法接受——例如，在存储卫星遥测数据时，或是用做关系数据库的内部格式时。

但即使是在这些情况下，用纯文本存储关于原始数据的元数据也可能是可以接受的（参见“元程序设计”，144页）。

有些开发者可能会担心，用纯文本存储元数据，是在把这些数据暴露给系统的用户。这种担心放错了地方。与纯文本相比，二进制数据也许更晦涩难懂，但却并非更安全。如果你担心用户看到密码，就进行加密。如果你不想让他们改变配置参数，就在文件中包含所有参数值的安全哈希值¹⁶作为校验和。

文本的威力

既然更大和更慢不是用户最想要的特性，为什么还要使用纯文本？好处是什么？

- 保证不过时
- 杠杆作用
- 更易于测试

保证不过时

人能够阅读的数据形式，以及自描述的数据，将比所有其他的数据形式和创建它们的应用都活得更长久。句号。

¹⁶ MD5 常被用于该目的。关于密码学奇妙世界的杰出引介，参见[Sch95]。

只要数据还存在，你就有机会使用它——也许是在原来创建它的应用已经不存在很久之后

只需部分地了解其格式，你就可以解析这样的文件；而对于大多数二进制文件，要成功地进行解析，你必须了解整个格式的所有细节

考虑一个来自某遗留系统¹⁷的数据文件。关于原来的应用你的了解很少；对你来说最要紧的是它保存了客户的社会保障号列表，你需要找出这些保障号，并将其提取出来。在数据文件中，你看到：

```
<FIELD10>123-45-6789</FIELD10>
...
<FIELD10>567-89-0123</FIELD10>
...
<FIELD10>901-23-4567</FIELD10>
```

识别出了社会保障号的格式，你可以很快写一个小程序提取该数据——即使你没有关于文件中其他任何东西的信息。

但设想一下，如果该文件的格式是这样的：

```
AC27123456789B11P
...
XY43567890123QTXL
...
6T2190123456788AM
```

你可能就不会那么轻松地识别出这些数字的含义了，这是人能够阅读（human readable）与人能够理解（human understandable）之间的区别

在我们进行解析时，**FIELD10** 的帮助也不大，改成

```
<SSNO>123-45-6789</SSNO>
```

就会让这个练习变得一点也不费脑子——而且这些数据保证会比创建它的任何项目都活得更长久

杠杆作用

实际上，计算世界中的每一样工具，从源码管理系统到编译器环境，再到编辑器及独立的过滤器，都能够在纯文本上进行操作

¹⁷ 所有软件一经写出，就成了“遗留的”。

Unix 哲学

提供“锋利”的小工具、其中每一样都意在把一件事情做好——Unix 因围绕这样的哲学进行设计而著称。这一哲学通过使用公共的底层格式得以实行：面向行的纯文本文件。用于系统管理（用户及密码、网络配置，等等）的数据库全都作为纯文本文件保存（有些系统，比如 Solaris，为了优化性能，还维护有特定数据的二进制形式。纯文本版本保留用作通往二进制版本的接口）。

当系统崩溃时，你可能需要通过最小限度的环境进行恢复（例如，你可能无法访问图形驱动程序）。像这样的情形，实在可以让你欣赏到纯文本的简单性。

例如，假定你要对一个大型应用进行产品部署，该应用具有复杂的针对具体现场的配置文件（我们想到 `sendmail`）。如果该文件是纯文本格式的，你可以把它置于源码控制系统的管理之下（参见源码控制，86 页），这样你就可以自动保存所有改动的历史。像 `diff` 和 `fc` 这样的文件比较工具允许你查看做了哪些改动，而 `sum` 允许你生成校验和，用以监视文件是否受到了偶然的（或恶意的）修改。

更易于测试

如果你用纯文本创建用于驱动系统测试的合成数据，那么增加、更新、或是修改测试数据就是一件简单的事情，而且无须为此创建任何特殊工具。与此类似，你可以非常轻松地分析回归测试（`regression test`）输出的纯文本，或通过 Perl、Python 及其他脚本工具进行更为全面彻底的检查。

最小公分母

即使在未来，基于 XML 的智能代理已能自治地穿越混乱、危险的 Internet、自行协商数据交换，无处不在的纯文本也仍然会存在。事实上，在异种环境中，纯文本的优点比其所有的缺点都重要。你需要确保所有各方能够使用公共标准进行通信。纯文本就是那个标准。

相关内容：

- 源码控制，86 页
- 代码生成器，102 页
- 元程序设计，144 页
- 黑板，165 页
- 无处不在的自动化，230 页
- 全都是写，248 页

挑战

- 使用你喜欢的语言，用直接的二进制表示设计一个小地址簿数据库（姓名、电话号码、等等）。完成以后再继续往下读。
 1. 把该格式转换成使用 XML 的纯文本格式。
 2. 在这两个版本中，增加一个新的、叫做方向的变长字段，在其中你可以输入每个人的住宅所在的方向。

在版本管理与可扩展性方面会遇到什么问题？哪种形式更易于修改？转换已有的数据呢？

15 shell 游戏

每个木匠都需要好用、坚固、可靠的工作台，用以在加工工件时把工件放置在方便的高度上。工作台成为木工房的中心，随着工件的成形，木匠会一次次回到工作台的近旁。

对于操纵文本文件的程序员，工作台就是命令 `shell`。在 `shell` 提示下，你可以调

用你的全套工具，并使用管道，以这些工具原来的开发者从未想过的方式把它们组合在一起。在 `shell` 下，你可以启动应用、调试器、浏览器、编辑器以及各种实用程序。你可以搜索文件、查询系统状态、过滤输出。通过对 `shell` 进行编程，你可以构建复杂的宏命令，用来完成你经常进行的各种活动。

对于在 GUI 界面和集成开发环境（IDE）上成长起来的程序员，这似乎显得很极端。毕竟，用鼠标指指点点，你不是也同样能把这些事情做好吗？

简单的回答：“不能”。GUI 界面很奇妙，对于某些简单操作，它们也可能更快、更方便。移动文件、阅读 MIME 编码的电子邮件以及写信，这都是你可能想要在图形环境中完成的事情。但如果你使用 GUI 完成所有的工作，你就会错过你的环境的某些能力。你将无法使常见任务自动化，或是利用各种可用工具的全部力量。同时，你也将无法组合你的各种工具，创建定制的宏工具。GUI 的好处是 WYSIWYG——所见即所得（`what you see is what you get`），缺点是 WYSIAYG——所见即全部所得（`what you see is all you get`）。

GUI 环境通常受限于它们的设计者想要提供的能力。如果你需要超越设计者提供的模型，你大概不会那么走运——而且很多时候，你确实需要超越这些模型。**注重实效的程序员**并非只是剪切代码、或是开发对象模型、或是撰写文档、或是使构建过程自动化——所有这些事情我们全都要做。通常，任何一样工具的适用范围都局限于该工具预期要完成的任务。例如，假定你需要把代码预处理器集成进你的 IDE 中（为了实现按合约设计、多处理编译指示，等等）。除非 IDE 的设计者明确地为这种能力提供了挂钩，否则，你无法做到这一点。

你也许已经习惯于在命令提示下工作，在这种情况下，你可以放心地跳过这一节。否则，你也许还需要我们向你证明，`shell` 是你的朋友。

作为**注重实效的程序员**，你不断地想要执行特别的操作——GUI 可能不支持的操作。当你想要快速地组合一些命令，以完成一次查询或某种其他的任务时，命令行要更为适宜。这里有一些例子：

找出修改日期比你的 **Makefile** 的修改日期更近的全部.c 文件。

Shell `find . -name '*.c' -newer Makefile -print`

GUI 打开资源管理器，转到正确的目录，点击 **Makefile**，记下修改时间。然后调出“工具/查找”，在指定文件处输入*.c。选择“日期”选项卡，在第一个日期字段中输入你记下的 **Makefile** 的日期。然后点击“确定”。

构造我的源码的 **zip/tar** 存档文件。

Shell `zip archive.zip *.h *.c` 或
`tar cvf archive.tar *.h *.c`

GUI 调出 ZIP 实用程序（比如共享软件 WinZip[URL 41]），选择[创建新存档文件]，输入它的名称，在“增加”对话框中选择源目录，把过滤器设置为“*.c”，点击“增加”，把过滤器设置为“*.h”，点击“增加”，然后关闭存档文件。

在上周哪些 **Java** 文件没有改动过？

Shell `find . -name '*.java' -mtime +7 -print`

GUI 点击并转到“查找文件”，点击“文件名”字段，敲入“*.java”，选择“修改日期”选项卡。然后选择“介于”点击“开始日期”，敲入项目开始的日期。点击“结束日期”，敲入 1 周以前的日期（确保手边有日历）。点击“开始查找”。

上面的文件中，哪些使用了 **awt** 库？

Shell `find . -name '*.java' -mtime +7 -print |`
`xargs grep 'java.awt'`

GUI 把前面的例子列出的各个文件装入编辑器，搜索字符串“Java.awt”。把含有该字符串的文件的名字写下来。

显然，这样的例子还可以一直举下去。**shell** 命令可能很晦涩，或是太简略，但却很强大，也很简练。同时，因为 **shell** 命令可被组合进脚本文件（或是 Windows 下的命令文件）中，你可以构建命令序列，使你常做的事情自动化。

提示 21**Use the Power of Command Shells****利用命令 shell 的力量**

去熟悉 shell，你会发现自己的生产率迅速提高。需要创建你的 Java 代码显式导入的全部软件包的列表（重复的只列出一次）？下面的命令将其存储在叫做“list”的文件中：

```
grep '^import ' *.java |
  sed -e's/.*import *//' -e's/;.*$//'|
  sort -u >list
```

如果你没有花大量时间研究过你所用系统上的命令 shell 的各种能力，这样的命令会显得很吓人。但是，投入一些精力去熟悉你的 shell，事情很快就会变得清楚起来。多使用你的命令 shell，你会惊讶它能使你的生产率得到怎样的提高。

shell 实用程序与 Windows 系统

尽管随 Windows 系统提供的命令 shell 在逐步改进，Windows 命令行实用程序仍然不如对应的 Unix 实用程序。但是，并非一切都已无可挽回。

Cygnus Solutions 公司有一个叫做 Cygwin[URL 31]的软件包。除了为 Windows 提供 Unix 兼容层以外，Cygwin 还带有 120 多个 Unix 实用程序，包括像 ls、grep 和 find 这样的很受欢迎的程序。你可以自由下载并使用这些实用程序和库，但一定要阅读它们的许可¹⁸。随同 Cygwin 发布的还有 Bash shell。

¹⁸ GNU General Public License[URL 57]是一种合法病毒，开放源码开发者将其用于保护他们的（及你的）权利。你应该花点时间阅读它。在本质上，它表明你可以使用并修改受 GPL 保护的软件，但如果你分发任何修改后的版本，必须采用 GPL 作为其许可方式（并作如是说明），同时你也必须公开源码。这就是病毒部分——只要你从受 GPL 保护的作品派生作品，你的派生作品也必须采用 GPL 许可。但是，如果你只是使用这些工具，它并不对这样的使用作任何限制——使用这些工具开发的软件的所有权及许可方式由你自行决定。

在 Windows 下使用 Unix 工具

在 Windows 下有高质量的 Unix 工具可用，这让我们很高兴；我们每天都使用它们。但是，要注意存在一些集成问题。与对应的 MS-DOS 工具不同，这些实用程序对文件名的大小写敏感，所以 `ls a*.bat` 不会找到 `AUTOEXEC.BAT`。你还可能遇到含有空格的文件名、或是路径分隔符不同所带来的问题。最后，在 Unix shell 下运行需要 MS-DOS 风格的参数的 MS-DOS 程序时，会发生一些有趣的问题。例如，在 Unix 下，来自 JavaSoft 的 Java 实用程序使用冒号作为 `CLASSPATH` 分隔符，而在 MS-DOS 下使用的却是分号。结果，运行在 Unix 机器上的 Bash 或 ksh 脚本在 Windows 下也同樣能运行，但它传给 Java 的命令行却会被错误地解释。

另外，David Korn（因 Korn shell 而闻名）制作了一个叫做 **UWIN** 的软件包。其目标与 Cygwin 相同——它是 Windows 下的 Unix 开发环境。UWIN 带有 Korn shell 的一个版本。也可从 Global Technologies, Ltd.[URL 30]获取商业版本。此外，AT&T 提供了该软件包的自由下载版本，用于评估和学术研究。再次说明，在使用之前要先阅读它们的许可。

最后，Tom Christiansen（在本书撰写的时候）正在制作 *Perl Power Tools*，尝试用 Perl 可移植地实现所有常见的 Unix 实用程序[URL 32]。

相关内容：

- 无处不在的自动化，230 页

挑战

- 你目前是否在 GUI 中用手工作一些事情？你是否曾将一些说明发给同事，其中涉及许多“点这个按钮”、“选哪一项”之类的步骤？它们能自动化吗？
- 每当你迁往新环境时，要找出可以使用的 shell。看是否能把现在使用的 shell 带过去。
- 调查各种可用于替换你现在的 shell 的选择。如果你遇到你的 shell 无法处理的问题，看其他 shell 是否能更好地应对。

16 强力编辑

先前我们说过，工具是手的延伸。噢，与任何其他软件工具相比，这都更适用于编辑器。你需要能尽可能不费力气地操纵文本，因为文本是编程的基本原材料。让我们来看一些能帮助你最大限度地利用编辑环境的一些常见特性和功能。

一种编辑器

我们认为你最好是精通一种编辑器，并将其用于所有编辑任务：代码、文档、备忘录、系统管理，等等。如果不坚持使用一种编辑器，你就可能会面临现代的巴别塔大混乱。你可能必须用每种语言的 IDE 内建的编辑器进行编码，用“all-in-one”办公软件编辑文档，或是用另一种内建的编辑器发送电子邮件。甚至你用于在 shell 中编辑命令行的键击都有可能不同¹⁹。如果你在每种环境中有不同的编辑约定和命令，要精通这些环境中的任何一种都会很困难。

你需要的是精通。只是依次输入，并使用鼠标进行剪贴是不够的。那样，在你的手中有了一个强大的编辑器，你却无法发挥出它的效能。敲击十次 `<` 或 `BACKSPACE`，把光标左移到行首，不会像敲击一次 `^A`、`Home` 或 `0` 那样高效。

提示 22

Use a Single Editor Well

用好一种编辑器

选一种编辑器，彻底了解它，并将其用于所有的编辑任务。如果你用一种编辑器（或一组键绑定）进行所有的文本编辑活动，你就不必停下来思考怎样完成文本操纵：必需的键击将成为本能反应。编辑器将成为你双手的延伸；键会在滑过文本和思想时

¹⁹ 在理想的情况下，你使用的 shell 使用的键绑定（keybinding）应该与你的编辑器使用的键绑定吻合。例如，Bash 既支持 vi 键绑定，也支持 emacs 键绑定。

歌唱起来。这就是我们的目标。

确保你选择的编辑器能在你使用的所有平台上使用。Emacs、vi、CRiSP、Brief 及其他一些编辑器可在多种平台上使用，并且常常既有 GUI 版本，也有非 GUI（文本屏幕）版本。

编辑器特性

除了你认为特别有用、使用时特别舒适的特性之外，还有一些基本能力，我们认为每个像样的编辑器都应该具备。如果你的编辑器缺少其中的任何能力，那么你或许就应该考虑换一种更高级的编辑器了。

- **可配置。**编辑器的所有方面都应该能按你的偏好（**preference**）配置，包括字体、颜色、窗口尺寸以及键击绑定（什么键执行什么命令）。对于常见的编辑操作，与鼠标或菜单驱动的命令相比，只使用键击效率更高，因为你的手无须离开键盘。
- **可扩展。**编辑器不应该只因为出现了新的编程语言就变得过时，它应该能集成你使用的任何编译器环境。你应该能把任何新语言或文本格式（XML、HTML 第 9 版，等等）的各种细微差别“教”给它。
- **可编程。**你应该能对编辑器编程，让它执行复杂的、多步骤的任务。可以通过宏或内建的脚本编程语言（例如，Emacs 使用了 Lisp 的一个变种）进行这样的编程。

此外，许多编辑器支持针对特定编程语言的特性，比如：

- 语法突显
- 自动完成
- 自动缩进
- 初始代码或文档样板
- 与帮助系统挂接
- 类 IDE 特性（编译、调试，等等）

像语法突显这样的特性听起来也许像是无关紧要的附加物，但实际上却可能非常有用，而且还能提高你的生产率。一旦你习惯了看到关键字以不同的颜色或字体出现，远在你启动编译器之前，没有以那样的方式出现的、敲错的关键字就会在你面前跳出来。

对于大型项目，能够在编辑器环境中进行编译、并直接转到出错处非常方便。Emacs 特别擅长进行这种方式的交互。

生产率

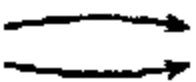
我们遇到的用 Windows notepad 编辑源码的人数量惊人。这就像是把茶匙当做铁锹——只是敲键和使用基本的基于鼠标的剪贴是不够的。

有什么样的事情需要你做，你却无法以这样的方式做到呢？

嗯，让我们以光标移动的例子作为开始。与重复击键、一个字符一个字符或一行一行移动相比，按一次键、就以词、行、块或函数为单位移动光标，效率要高得多。

再假设你在编写 Java 代码。你想要按字母顺序排列 import 语句，而另外有人签入（check in）了一些文件，没有遵守这一标准（这听起来也许很极端，但在大型项目中，这可以让你节省大量时间，不用逐行检查一大堆 import 语句）。你想要快速地从头到尾检查一些文件，并对它们的一小部分区域进行排序。在像 vi 和 Emacs 这样的编辑器中，你可以很容易完成这样的任务（参见图 3.1）。用 notepad 试试看！

图 3.1 在编辑器中对文本行进行排序

<code>import java.util.Vector;</code>	<code>emacs: M-x sort-lines</code>	<code>import java.awt.*;</code>
<code>import java.util.Stack;</code>		<code>import java.net.URL;</code>
<code>import java.net.URL;</code>		<code>import java.util.Stack;</code>
<code>import java.awt.*;</code>	<code>vi: :.,+3!sort</code>	<code>import java.util.Vector;</code>

有些编辑器能帮助你使常用操作流水线化。例如，当你创建特定语言的新文件时，编辑器可以为你提供模板，其中也许包括：

- 填好的类名或模块名（根据文件名派生）
- 你的姓名和/或版权声明
- 该语言中的各种构造体（construct）的骨架（例如，构造器与析构器声明）

自动缩进是另一种有用的特性。你不必（使用空格或 `tab`）进行手工缩进，编辑器会自动在适当的时候（例如，在敲入左花括号时）为你进行缩进。这一特性让人愉快的地方是，你可以用编辑器为你的项目提供一致的缩进风格²⁰

然后做什么

这种建议特别难写，因为实际上每个读者对他们所用编辑器的熟悉程度和相关经验都有所不同。那么，作为总结，并为下一步该做什么提出一些指导方针，在下面的左边一栏中找到与你的情况相符的情况，然后看右边一栏，看你应该做什么。

如果这听起来像你……

那么考虑……

我使用许多不同的编辑器，但只使用其基本特性。

选一种强大的编辑器，好好学习它。

我有最喜欢的编辑器，但不使用其全部特性。

学习它们，减少你需要敲击的键数。

我有最喜欢的编辑器，只要可能就使用它。

设法扩展它，并将其用于比现在更多的任务。

我认为你们在胡说。notepad 就是有史以来最好的编辑器。

只要你愿意，并且生产率很高，那就这样吧！但如果你发现自己在“羡慕”别人的编辑器，你可能就需要重新评估自己的位置了。

²⁰ Linux 内核就是以这种方式开发的。你拥有在地理上散布各处的开发者，有许多人在同一段代码上工作。有一份公开的设置列表（在本例中，是用于 Emacs 的），描述了所需的缩进风格。

有哪些编辑器可用

此前我们建议你掌握一种像样的编辑器，那么我们推荐哪种编辑器呢？嗯，我们要回避这个问题：你对编辑器的选择是一个个人问题（有人甚至会说这是个“信仰问题”！）但是，在附录 A（266 页）中，我们列出了许多流行的编辑器和获取它们的途径。

挑战

- 有些编辑器使用完备的语言进行定制和脚本编写。例如，Emacs 采用了 Lisp。作为本年度你将学习的新语言之一，学习你的编辑器使用的语言。如果你发现自己在重复做任何事情，开发一套宏（或等价的东西）加以处理。
- 你是否知道你的编辑器所能做的每一件事情？设法难倒使用同样的编辑器的同事。设法通过尽可能少的键击完成任何给定的编辑任务。

17 源码控制

进步远非由变化组成，而是取决于好记性。不能记住过去的人，被判重复过去。

——George Santayana, *Life of Reason*

我们在用户界面中找寻的一个重要的东西是 **UNDO** 键——一个能原谅我们的错误的按钮。如果环境支持多级撤消（undo）与重做（redo），那就更好了，这样你就可以回去，撤消几分钟前发生的事情。但如果错误发生在上周，而你那以后已经把计算机打开关闭了十次呢？噢，这是使用源码控制系统的诸多好处之一：它是一个巨大的 **UNDO** 键——一个项目级的时间机器，能够让你返回上周的那些太平日子，那时的代码还能够编译并运行。

源码控制系统（或范围更宽泛的配置管理系统）追踪你在源码和文档中做出的每一项变动。

更好的系统还能追踪编译器及 OS 版本。有了适当配置的源码控制系统，你就总能够返回你的软件的前一版本。

但源码控制系统（SCCS²¹）能做的远比撤消错误要多。好的 SCCS 让你追踪变动，回答这样的问题：谁改动了这一行代码？在当前版本与上周的版本之间有什么区别？在这次发布的版本中我们改动了多少行代码？哪个文件改动最频繁？对于 bug 追踪、审计、性能及质量等目的，这种信息非常宝贵。

SCCS 还能让你标识你的软件的各次发布。一经标识，你将总是能够返回并重新生成该版本，并且不受在其后发生的变动的影响。

我们常常使用 SCCS 管理开发树中的分支。例如，一旦你发布了某个软件，你通常会想为下一次发布继续开发。与此同时，你也需要处理当前发布的版本中的 bug，把修正后的版本发送给客户。（如果合适）你想要让这些 bug 修正合并进下一次发布中，但你不想把正在开发的代码发送给客户。通过 SCCS，在每次生成一个发布版本时，你可以在开发树中生成分支。你把 bug 修正加到分支中的代码上，并在主干上继续开发。因为 bug 修正也可能与主干有关，有些系统允许你把选定的来自分支的变动自动合并回主干中。

源码控制系统可能会把它们维护的文件保存在某个中央仓库（repository）中——这是进行存档的好候选地。

最后，有些产品可能允许两个或更多用户同时在相同的文件集上工作，甚至在同一文件中同时做出改动。系统随后在文件被送回仓库时对这些改动进行合并。尽管看起来有风险，在实践中这样的系统在所有规模的项目上都工作良好。

²¹ 我们使用大写的 SCCS 指称一般的源码控制系统。还有一个叫做“sccs”的具体系统，该系统最初随 AT&T System V Unix 一起发布。

提示 23**Always Use Source Code Control****总是使用源码控制**

总是。即使你的团队只有你一个人，你的项目只需一周时间；即使那是“用过就扔”的原型；即使你的工作对象并非源码；确保每样东西都处在源码控制之下——文档、电话号码表、给供应商的备忘录、makefile、构建与发布流程、烧制 CD 母盘的 shell 小脚本——每样东西。我们例行公事地对我们敲入的每一样东西进行源码控制（包括本书的文本）。即使我们不是在开发项目，我们的日常工作也被安全地保存在仓库中。

源码控制与构建

把整个项目置于源码控制系统的保护之下具有一项很大的、隐蔽的好处：你可以进行自动的和可重复的产品构建。

项目构建机制可以自动从仓库中取出最近的源码。它可以在午夜运行，在每个人都（很可能）回家之后。你可以运行自动的回归测试，确保当日的编码没有造成任何破坏。构建的自动化保证了一致性——没有手工过程，而你也不需要开发者记住把代码拷贝进特殊的构建区域。

构建是可重复的，因为你总是可以按照源码将给定日期的内容重新进行构建。

但我们团队没有使用源码控制

他们应该感到羞耻！听起来这是个“布道”的机会！但是，在等待他们看到光明的同时，也许你应该实施自己私人的源码控制。使用我们在附录 A 中列出的可自由获取的工具，并确保把你个人的工作安全地保存进仓库中（并且完成你的项目所要求的无论什么事情）。尽管这看起来像是重复劳动，我们几乎可以向你担保，在你须要回答

像“你对 xyz 模块做了什么？”和“是什么破坏了构建？”这样的问题时，它将使你免受困扰（并为你的项目节省金钱）这一方法也许还能有助于使你们的管理部门确信，源码控制确实行之有效

不要忘了，SCCS 也同样适用于你在工作之外所做的事情

源码控制产品

附录 A（271 页）给出了一些有代表性的源码控制系统的 URL，有些是商业产品，有些可自由获取——还有许多其他的产品可用——你可以在配置管理 FAQ 中寻求建议

相关内容：

- 正交性，34 页
- 纯文本的力量，73 页
- 全都是写，248 页

挑战

- 即使你无法在工作中使用 SCCS，也要在个人的系统上安装 RCS 或 CVS——用它管理你的“宠物项目”、你撰写的文档、以及（可能的）应用于计算机系统自身的配置变动。
- 在 Web 上有些开放源码项目的存档对外公开（比如 Mozilla[URL51]、KDE[URL54]、以及 Gimp[URL55]），看一看这样的项目。你怎样获取源文件的更新？你怎样做出改动？——项目是否会对访问进行管制，或是对改动的并入进行裁决？

18 调试

这是痛苦的事：

看着你自己的烦恼，并且知道

不是别人，而是你自己一人所致

——索福克勒斯：《埃阿斯》

自从 14 世纪以来，*bug*（虫子、臭虫）一词就一直被用于描述“恐怖的东西”。COBOL 的发明者、海军少将 Grace Hopper 博士据信观察到了第一只计算机 *bug*——真的是一只虫子，一只在早期计算机系统的继电器里抓到的蛾子。在被要求解释机器为何未按期望运转时，有一位技术人员报告说，“有一只虫子在系统里”，并且负责地把它——翅膀及其他所有部分——粘在了日志簿里。

遗憾的是，在我们的系统里仍然有“*bug*”，虽然不是会飞的那种。但与以前相比，14 世纪的含义——可怕的东西——现在也许更为适用。软件缺陷以各种各样的方式表现自己，从被误解的需求到编码错误。糟糕的是，现代计算机系统仍然局限于做你告诉它的事情，而不一定是你想要它做的事情。

没有人能写出完美的软件，所以调试肯定要占用你大量时间。让我们来看一看调试所涉及的一些问题，以及一些用于找出难以捉摸的虫子的一般策略。

调试的心理学

对于许多开发者，调试本身是一个敏感、感性的话题。你可能会遇到抵赖、推诿、蹩脚的借口、其或是无动于衷，而不是把它当做要解决的难题发起进攻。

要接受事实：调试就是解决问题，要据此发起进攻。

发现了他人的 *bug* 之后，你可以花费时间和精力去指责让人厌恶的肇事者。在有些工作环境中，这是文化的一部分，并且可能是“疏通剂”。但是，在技术竞技场上，你应该专注于修正问题，而不是发出指责。

提示 24

Fix the Problem, Not the Blame

要修正问题，而不是发出指责

bug 是你的过错还是别人的过错，并不是真的很有关系。它仍然是你的问题。

调试的思维方式

最容易欺骗的人是一个人自己。

——Edward Bulwer-Lytton, *The Disowned*

在你开始调试之前，选择恰当的思维方式十分重要。你须要关闭每天用于保护自我（ego）的许多防卫措施，忘掉你可能面临的任何项目压力，并让自己放松下来。最重要的是，记住调试的第一准则：

提示 25

Don't Panic

不要恐慌

人很容易恐慌，特别是如果你正面临最后期限的到来，或是正在设法找出 bug 的原因，有一个神经质的老板或客户在你的脖子后面喘气。但非常重要的事情是，要后退一步，实际思考什么可能造成你认为表征了 bug 的那些症状。

如果你目睹 bug 或见到 bug 报告时的第一反应是“那不可能”，你就完全错了。一个脑细胞都不要浪费在以“但那不可能发生”起头的思路，因为很明显，那不仅可能，而且已经发生了。

在调试时小心“近视”。要抵制只修正你看到的症状的急迫愿望；更有可能的情况是，实际的故障离你正在观察的地方可能还有几步远，并且可能涉及许多其他的相关事物。要总是设法找出问题的根源，而不只是问题的特定表现。

从何处开始

在开始查看 bug 之前，要确保你是在能够成功编译的代码上工作——没有警告。

我们例行公事地把编译器警告级设得尽可能高。把时间浪费在设法找出编译器能够为你找出的问题上没有意义！我们需要专注于手上更困难的问题。

在设法解决任何问题时，你需要搜集所有的相关数据。糟糕的是，bug 报告不是精密科学。你很容易被巧合误导，而你不能承受把时间浪费在对巧合进行调试上。你首先需要在观察中做到准确。

bug 报告的准确性在经过第三方之手时会进一步降低——实际上你可能需要观察报告 bug 的用户的操作，以获取足够程度的细节。

Andy 曾经参与过一个大型图形应用的开发。快要发布时，测试人员报告说，每次他们用特定的画笔画线，应用都会崩溃。负责该应用的程序员争辩说，这个画笔没有任何问题；他试过用它绘图，它工作得很好。几天里这样的对话来回进行，大家的情绪急速上升。

最后，我们让他们坐到同一个房间里。测试人员选了画笔工具，从右上角到左下角画了一条线。应用程序炸了。“噢”，程序员用很小的声音说。他随后像绵羊一样承认，他在测试时只测试了从左下角画到右上角的情况，没有暴露出这个 bug。

这个故事有两个要点：

- 你也许需要与报告 bug 的用户面谈，以搜集比最初给你的数据更多的数据。
- 人工合成的测试（比如那个程序员只从下画到上）不能足够地演练（exercise）应用。你必须既强硬地测试边界条件，又测试现实中的最终用户的使用模式。你需要系统地进行这样的测试（参见无情的测试，237 页）。

测试策略

一旦你认为你知道了在发生什么，就到了找出程序认为在发生什么的时候了。

再现 bug (reproduction, 亦有“繁殖”之意——译注)

不,我们的 bug 不会真的繁殖(尽管其中有一些可能已经到了合法的生育年龄) 我们谈论的是另一种“再现”

开始修正 bug 的最佳途径是让其可再现 毕竟,如果你不能再现它,你又怎么知道它已经被修正了呢?

但我们想要的不是能够通过长长的步骤再现的 bug;我们要的是能够通过一条命令再现的 bug。如果你必须通过 15 个步骤才能到达 bug 显露的地方,修正 bug 就会困难得多 有时候,强迫你自己隔离显示出 bug 的环境,你甚至会洞见到它的修正方法

要了解沿着这些思路延伸的其他想法,参见无处不在的自动化(230 页)

使你的数据可视化

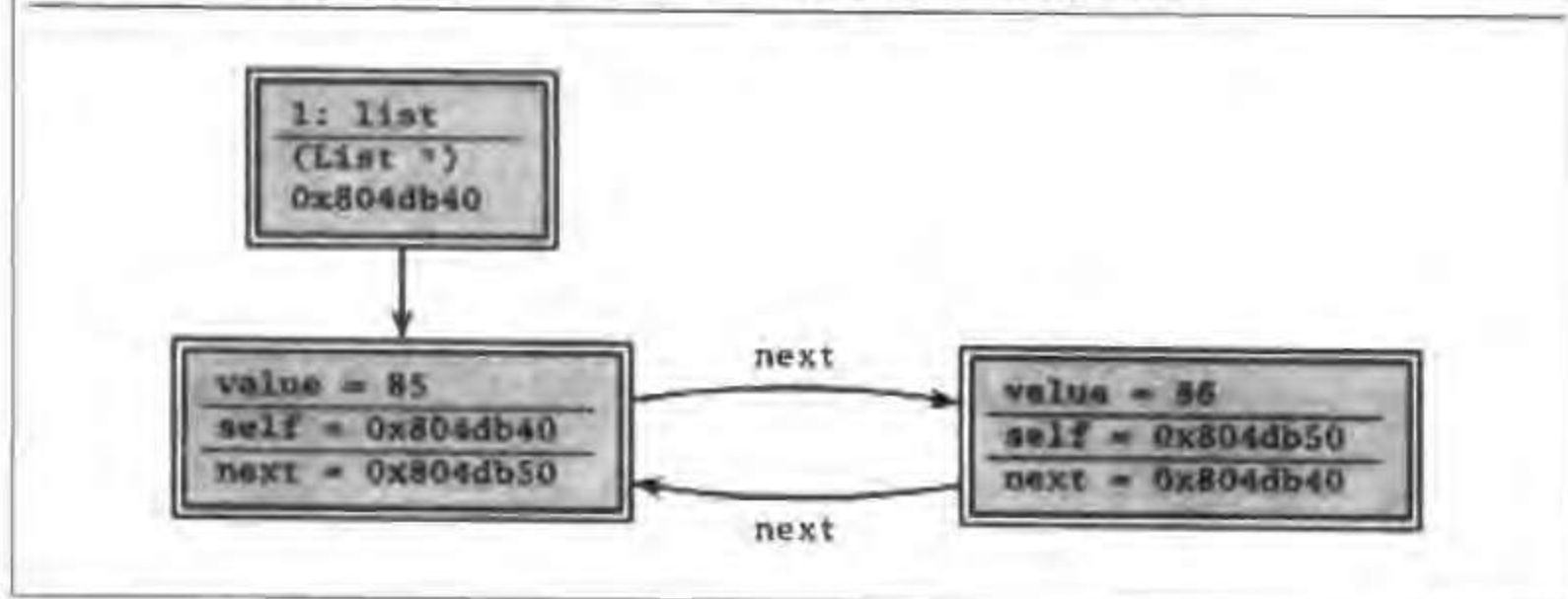
常常,要认识程序在做什么——或是要做什么——最容易的途径是好好看一看它操作的数据 最简单的例子是直截了当的“variable name = data value”方法,这可以作为打印文本、也可以作为 GUI 对话框或列表中的字段实现

但通过使用允许你“使数据及其所有的相互关系可视化”的调试器,你可以深入得多地获得对你的数据的洞察。有一些调试器能够通过虚拟现实场景把你的数据表示为 3D 立交图,或是表示为 3D 波形图,或是就表示为简单的结构图(如下--页的图 3.2 所示)。在单步跟踪程序的过程中,当你一直在追猎的 bug 突然跳到你面前时,这样的图远胜于千言万语。

即使你的调试器对可视化数据的支持有限,你仍然自己进行可视化——或是通过手工方式,用纸和笔,或是用外部的绘图程序。

DDD 调试器有一些可视化能力,并且可以自由获取(参见[URL 19])。有趣的是,DDD 能与多种语言一起工作,包括 Ada、C、C++、Fortran、Java、Modula、Pascal、

图 3.2 一个循环链表的调试器示例图。箭头表示指向节点的指针



Perl 以及 Python（显然是正交的设计）。

跟踪

调试器通常会聚焦于程序现在的状态。有时你需要更多的东西——你需要观察程序或数据结构随时间变化的状态。查看栈踪迹（stack trace）只能告诉你，你是怎样直接到达这里的。它无法告诉你，在此调用链之前你在做什么，特别是在基于事件的系统中。

跟踪语句把小诊断消息打印到屏幕上或文件中，说明像“到了这里”和“x 的值 = 2”这样的事情。与 IDE 风格的调试器相比，这是一种原始的技术，但在诊断调试器无法诊断的一些错误种类时却特别有效。在时间本身是一项因素的任何系统中，跟踪都具有难以估量的价值：并发进程、实时系统、还有基于事件的应用。

你可以使用跟踪语句“钻入”代码。也就是，你可以在沿着调用树下降时增加跟踪语句。

跟踪消息应该采用规范、一致的格式：你可能会想自动解析它们。例如，如果你需要跟踪资源泄漏（比如未配平（unbalanced）的 open/close），你可以把每一次 open 和每一次 close 记录在日志文件中。通过用 Perl 处理该日志文件，你可以轻松地确定

坏变量？检查它们的邻居

有时你检查一个变量，希望看到一个小整数值，得到的却是像 0x6e69614d 这样的东西。在你卷起袖子、郑重其事地开始调试之前，先快速地查看一下这个坏变量周围的内存。这常常能带给你线索。在我们的例子中，把周边的内存作为字符进行检查得到的是：

```
20333231 6e69614d 2c745320 746f4e0a
1 2 3   M a i n       S t , \n N o t
      2c6e776f 2058580a 31323433 00000a33
      o w n , \n x x   3 4 2 1 3 \n \0 \0
```

看上去像是有人把街道地址“喷”到了我们的计数器上。现在我们知道该去查看什么地方了。

有问题的 open 是在哪里发生的。

橡皮鸭

找到问题的原因的一种非常简单、却又特别有用的技术是向别人解释它。他应该越过你的肩膀看着屏幕，不断点头（像澡盆里上下晃动的橡皮鸭）。他们一个字也不需要说；你只是一步步解释代码要做什么，常常就能让问题从屏幕上跳出来，宣布自己的存在²²。

这听起来很简单，但在向他人解释问题时，你必须明确地陈述那些你在自己检查代码时想当然的事情。因为必须详细描述这些假定中的一部分，你可能会突然获得对新洞见。

²² 为什么说“橡皮鸭”？在伦敦皇家学院学习时，Dave 曾经和一位叫做 Greg Pugh 的研究助理一起做过大量工作，他是 Dave 认识的最好的开发者之一。有几个月 Greg 带了一只黄色的小鸭子，在编码时把它放在他的终端上。过了一阵子，Dave 才鼓起勇气问……

消除过程

在大多数项目中，你调试的代码可能是你和你们团队的其他成员编写的应用代码、第三方产品（数据库、连接性、图形库、专用通信或算法，等等），以及平台环境（操作系统、系统库、编译器）的混合物。

bug 有可能存在于 OS、编译器、或是第三方产品中——但这不应该是你的第一想法。有大得多的可能性的是，bug 存在于正在开发的应用代码中。与假定库本身出了问题相比，假定应用代码对库的调用不正确通常更有好处。即使问题确实应归于第三方，在提交 bug 报告之前，你也必须先消除你的代码中的 bug。

我们参加过一个项目的开发，有位高级工程师确信 select 系统调用在 Solaris 上有问题。再多的劝说或逻辑也无法改变他的想法（这台机器上的所有其他网络应用都工作良好这一事实也一样无济于事）。他花了数周时间编写绕开这一问题的代码，因为某种奇怪的原因，却好像并没有解决问题。当最后被迫坐下来、阅读关于 select 的文档时，他在几分钟之内就发现并纠正了问题。现在每当有人开始因为很可能是我们自己的故障而抱怨系统时，我们就会使用“select 没有问题”作为温和的提醒。

提示 26

“Select” Isn’t Broken

“Select” 没有问题

记住，如果你看到马蹄印，要想到马，而不是斑马。OS 很可能没有问题。数据库也很可能情况良好。

如果你“只改动了一样东西”，系统就停止了工作，那样东西很可能就需要对此负责——直接地或间接地，不管那看起来有多牵强。有时被改动的东西在你的控制之外：OS 的新版本、编译器、数据库或是其他第三方软件都可能会毁坏先前的正确代码。可能会出现新的 bug。你先前已绕开的 bug 得到了修正，却破坏了用于绕开它的代码。API 变了，功能变了；简而言之，这是全新的球赛，你必须在这些新的条件下重新测

试系统。所以在考虑升级时要紧盯着进度表：你可能会想等到下一次发布之后再升级。

但是，如果没有显而易见的地方让你着手查看，你总是可以依靠好用的老式二分查找。看症状是否出现在代码中的两个远端之一，然后看中间。如果问题出现了，则臭虫位于起点与中点之间；否则，它就在中点与终点之间。以这种方式，你可以让范围越来越小，直到最终确定问题所在。

造成惊讶的要素

在发现某个 bug 让你吃惊时（也许你在用我们听不到的声音咕哝说：“那不可能。”），你必须重新评估你确信不疑的“事实”。在那个链表例程中——你知道它坚固耐用，不可能是这个 bug 的原因——你是否测试了所有边界条件？另外一段代码你已经用了好几年——它不可能还有 bug。可能吗？

当然可能。某样东西出错时，你感到吃惊的程度与你对正在运行的代码的信任及信心成正比。这就是为什么，在面对“让人吃惊”的故障时，你必须意识到你的一个或更多的假设是错的。不要因为你“知道”它能工作而轻易放过与 bug 有牵连的例程或代码。证明它。用这些数据、这些边界条件，在这个语境中证明它。

提示 27

Don't Assume it – Prove It

不要假定，要证明

当你遇到让人吃惊的 bug 时，除了只是修正它而外，你还需要确定先前为什么没有找出这个故障。考虑你是否需要改进单元测试或其他测试，以让它们有能力找出这个故障。

还有，如果 bug 是一些坏数据的结果，这些数据在造成爆发之前传播通过了若干层面，看一看在这些例程中进行更好的参数检查是否能更早地隔离它（分别参见 120 页与 122 页的关于早崩溃及断言的讨论）。

在你对其进行处理的同时，代码中是否有任何其他地方容易受这同一个 bug 的影响？现在就是找出并修正它们的时机。确保无论发生什么，你都知道它是否会再次发生。

如果修正这个 bug 需要很长时间，问问你自己为什么。你是否可以做点什么，让下一次修正这个 bug 变得更容易？也许你可以内建更好的测试挂钩，或是编写日志文件分析器。

最后，如果 bug 是某人的错误假定的结果，与整个团队一起讨论这个问题。如果一个人有误解，那么许多人可能也有。

去做所有这些事情，下一次你就将很有希望不再吃惊。

调试检查列表

- 正在报告的问题是底层 bug 的直接结果，还是只是症状？
- bug 真的在编译器里？在 OS 里？或者是在你的代码里？
- 如果你向同事详细解释这个问题，你会说什么？
- 如果可疑代码通过了单元测试，测试是否足够完整？如果你用该数据运行单元测试，会发生什么？
- 造成这个 bug 的条件是否存在于系统中的其他任何地方？

相关内容：

- 断言式编程，122 页
- 靠巧合编程，172 页
- 无处不在的自动化，230 页
- 无情的测试，237 页

挑战

- 调试已经够有挑战性了。

19 文本操纵

注重实效的程序员用与木匠加工木料相同的方式操纵文本。在前面的部分里，我们讨论了我们所用的一些具体工具——**shell**、编辑器、调试器。这些工具与木匠的凿子、锯子、刨子类似——它们都是用于把一件或两件工作做好的专用工具。但是，我们不时也需要完成一些转换，这些转换不能由基本工具集直接完成。我们需要通用的文本操纵工具。

文本操纵语言对于编程的意义，就像是刨刨机（**router**）²³对于木工活的意义。它们嘈杂、肮脏、而且有点用“蛮力”。如果使用有误，整个工件都可能毁坏。有人发誓说在工具箱里没有它们的位置。但在恰当的人的手中，刨刨机和文本操纵语言都可以让人难以置信地强大和用途广泛。你可以很快把某样东西加工成形、制作接头、并进行雕刻。如果适当使用，这些工具拥有让人惊讶的精微与巧妙。但你需要花时间才能掌握它们。

好的文本操纵语言的数目正在增长。Unix 开发者常常喜欢利用他们的命令 **shell** 的力量，并用像 **awk** 和 **sed** 这样的工具加以增强。偏爱更为结构化的工具的人喜欢 **Python**[URL 9]的面向对象本质。有人把 **Tcl**[URL 23]当作自己的首选工具。我们碰巧喜欢用 **Perl**[URL 8]编写短小的脚本。

这些语言是能赋予你能力的重要技术。使用它们，你可以快速地构建实用程序，为你的想法建立原型——使用传统语言，这些工作可能需要 5 倍或 10 倍的时间。对于我们所做的实验，这样的放大系数十分重要。与花费 5 小时相比，花费 30 分钟试验一个疯狂的想法要好得多。花费 1 天使项目的重要组件自动化是可以接受的；花费 1 周却不一定。在 *The Practice of Programming*[KP99]一书中，Kernighan 与 Pike 用 5 种不同的语言构建同一个程序。**Perl** 版本是最短的（17 行，而 C 要 150 行）。通过 **Perl** 你可以操纵文本、与程序交互、进行网络通信、驱动网页、进行任意精度的运算、以

²³ 这里的 **router** 指的是非常、非常快地旋转切割刀片的工具，不是用于网络互连的设备

及编写看起来像史努比发誓的程序。

提示 28

Learn a Text Manipulation Language

学习一种文本操纵语言

为了说明文本操纵语言的广泛适用性，这里列出了我们过去几年开发的一些应用示例：

- **数据库 schema 维护。**一组 Perl 脚本读取含有数据库 schema 定义的纯文本文件，根据它生成：
 - 用于创建数据库的 SQL 语句
 - 用于填充数据词典的平板（flat）数据文件
 - 用于访问数据库的 C 代码库
 - 用于检查数据库完整性的脚本
 - 含有 schema 描述及框图的网页
 - schema 的 XML 版本
- **Java 属性（property）访问。**限制对某个对象的属性的访问，迫使外部类通过方法获取和设置它们，这是一种良好的 OO 编程风格。但是，属性在类的内部由简单的成员变量表示是一种常见情况，在这样的情况下要为每个变量创建获取和设置方法既乏味，又机械。我们有一个 Perl 脚本，它修改源文件，为所有做了适当标记的变量插入正确的方法定义。
- **测试数据生成。**我们的测试数据有好几万记录，散布在若干不同的文件中，其格式也不同，它们需要汇合在一起，并转换为适于装载进关系数据库的某种形式。Perl 用几小时就完成了这一工作（在此过程中还发现了初始数据的几处一致性错误）。

- **写书。**我们认为，出现在书籍中的任何代码都应首先进行测试，这十分重要。本书中的大多数代码都经过了测试。但是，按照 *DRY* 原则（参见“重复的危害”，26 页），我们不想把代码从测试过的程序拷贝并粘贴到书里。那意味着代码是重复的，实际上我们肯定会在程序被改动时忘记更新相应的例子。对于有些例子，我们也不想用编译并运行例子所需的全部框架代码来烦扰你。我们转向了 Perl。在我们对书进行格式化时，会调用一个相对简单的脚本——它提取源文件中指定的片段，进行语法突显，并把结果转换成我们使用的排版语言。
- **C 与 Object Pascal 的接口。**某个客户有一个在 PC 上编写 Object Pascal 应用的开发团队。他们的代码需要与用 C 编写的一段代码接口。我们开发了一个短小的 Perl 脚本，解析 C 头文件，提取所有被导出函数的定义，以及它们使用的数据结构。随后我们生成 Object Pascal 单元：用 Pascal 记录对应所有的 C 结构，用导入的过程定义对应所有的 C 函数。这一生成过程变成了构建的一部分，这样无论何时 C 头文件发生变化，新的 Object Pascal 单元都会自动被构造。
- **生成 Web 文档。**许多项目团队都把文档发布在内部网站上。我们编写了许多 Perl 程序，分析数据库 schema、C 或 C++ 源文件、makefile 以及其他项目资源，以生成所需的 HTML 文档。我们还使用 Perl，把文档用标准的页眉和页脚包装起来，并把它们传输到网站上。

我们几乎每天都使用文本操纵语言。与我们注意到的其他任何语言相比，本书中的许多想法都可以用这些语言更简单地实现。这些语言使我们能够轻松地编写代码生成器，我们将在下一节讨论这一主题。

相关内容：

- 重复的危害，26 页

练习

11. 你的 C 程序使用枚举类型表示 100 种状态。为进行调试，你想要能把状态打印成（与数字对应的）字符串。编写一个脚本，从标准输入读取含有以下内容的文件：
（解答在 285 页）

```
name
state_a
state_b
:      :
```

生成文件 *name.h*，其中含有：

```
extern const char* NAME_names[];
typedef enum {
    state_a,
    state_b,
    :      :
} NAME;
```

以及文件 *name.c*，其中含有：

```
const char* NAME_names[] = {
    "state_a",
    "state_b",
    :      :
};
```

12. 在本书撰写中途，我们意识到我们没有把 `use strict` 指示放进我们的许多 Perl 例子。编写一个脚本，检查某个目录中的 .pl 文件，给没有 `use strict` 指示的所有文件在初始注释块的末尾加上该指示。要记住给你改动的所有文件保留备份。（解答在 286 页）

20 代码生成器

当木匠面临一再地重复制作同一样东西的任务时，他们会取巧。他们给自己建造夹具或模板。一旦他们做好了夹具，他们就可以反复制作某样工件，夹具带走了复杂性，降低了出错的机会，从而让工匠能够自由地专注于质量问题。

作为程序员，我们常常发现自己也处在同样的位置上。我们需要获得同一种功能，

但却是在不同的语境中。我们需要在不同的地方重复信息。有时我们只是需要通过减少重复的打字，使自己免于患上腕部劳损综合症。

以与木匠在夹具上投入时间相同的方式，程序员可以构建代码生成器。一旦构建好，在整个项目生命期内都可以使用它，实际上没有任何代价。

提示 29

Write Code That Writes Code

编写能编写代码的代码

代码生成器有两种主要类型：

1. 被动代码生成器只运行一次来生成结果。然后结果就变成了独立的——它与代码生成器分离了。在 198 页的邪恶的向导中讨论的向导，还有某些 CASE 工具，都是被动代码生成器的例子。
2. 主动代码生成器在每次需要其结果时被使用。结果是用过就扔的——它总是能由代码生成器重新生成。主动代码生成器为了生成其结果，常常要读取某种形式的脚本或控制文件。

被动代码生成器

被动代码生成器减少敲键次数。它们本质上是参数化模板，根据一组输入生成给定的输出形式。结果一经产生，就变成了项目中有充分资格的源文件；它将像任何其他文件一样被编辑、编译、置于源码控制之下。其来源将被忘记。

被动代码生成器有许多用途：

- 创建新的源文件。被动代码生成器可以生成模板、源码控制指示、版权说明以及项目中每个新文件的标准注释块。我们设置我们的编辑器，让它在我们每次创建新文件时做这样的工作：编辑新的 Java 程序，新的编辑器缓冲区将自动包含注释块、包指示以及已经填好的概要的类声明。

- 在编程语言之间进行一次性转换。我们开始撰写本书时使用的是 `troff` 系统，但我们在完成了 15 节以后转向了 `LaTeX`。我们编写了一个代码生成器，读取 `troff` 源，并将其转换到 `LaTeX`。其准确率大约是 90%，余下部分我们用手工完成。这是被动代码生成器的一个有趣的特性：它们不必完全准确。你需要在你投入生成器的努力和你花在修正其输出上的精力之间进行权衡。
- 生成查找表及其他在运行时计算很昂贵的资源。许多早期的图形系统都使用预先计算的正弦和余弦值表，而不是在运行时计算三角函数。在典型情况下，这些表由被动代码生成器生成，然后拷贝到源文件中。

主动代码生成器

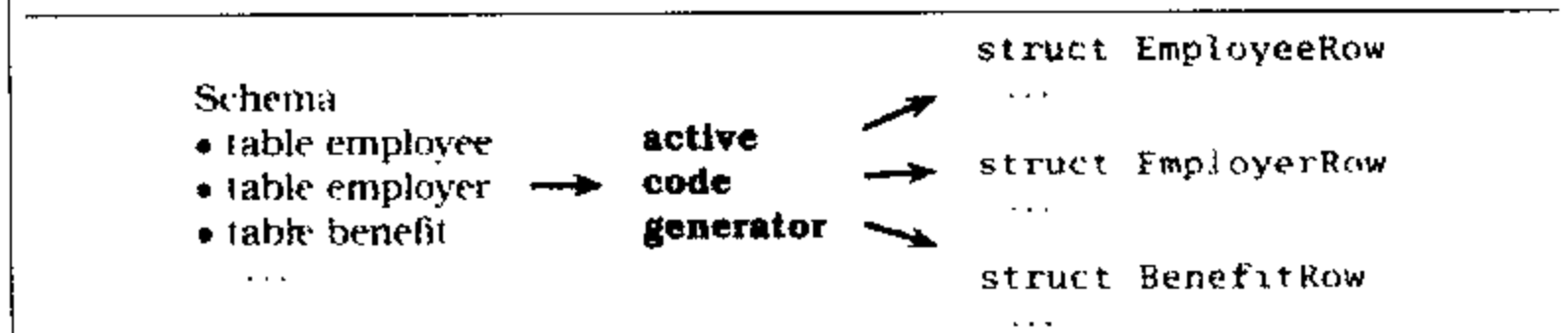
被动代码生成器只是一种便利手段，如果你想要遵循 *DRY* 原则，它们的“表亲”主动代码生成器却是必需品。通过主动代码生成器，你可以取某项知识的一种表示形式，将其转换为你的应用需要的所有形式。这不是重复，因为衍生出的形式可以用过就扔，并且是由代码生成器按需生成的（所以才会用主动这个词）。

无论何时你发现自己在设法让两种完全不同的环境一起工作，你都应该考虑使用主动代码生成器。

或许你在开发数据库应用。这里，你在处理两种环境——数据库和你用来访问它的编程语言。你有一个 `schema`，你需要定义低级的结构，反映特定的数据库表的布局。你当然可以直接对其进行编码，但这违反了 *DRY* 原则：`schema` 的知识就会在两个地方表示。当 `schema` 变化时，你需要记住改变相应的代码。如果某一行从表中被移走，而代码库却没有改变，甚至有可能连编译错误也没有。只有等你的测试开始失败时（或是用户打电话过来），你才会知道它。

另一种办法是使用主动代码生成器——如图 3.3 所示，读取 `schema`，使用它生成结构的源码。现在，无论何时 `schema` 发生变化，用于访问它的代码也会自动变化。如果某一行被移走，那么它在结构中相应的字段也将消失，任何使用该列的更高级的代码就将无法通过编译。

图 3.3 主动代码生成器根据数据库 schema 创建代码



你在编译时就能抓住错误，不用等到投入实际运行时。当然，只有在你让代码生成成为构建过程自身的一部分的情况下，这个方案才能工作²⁴

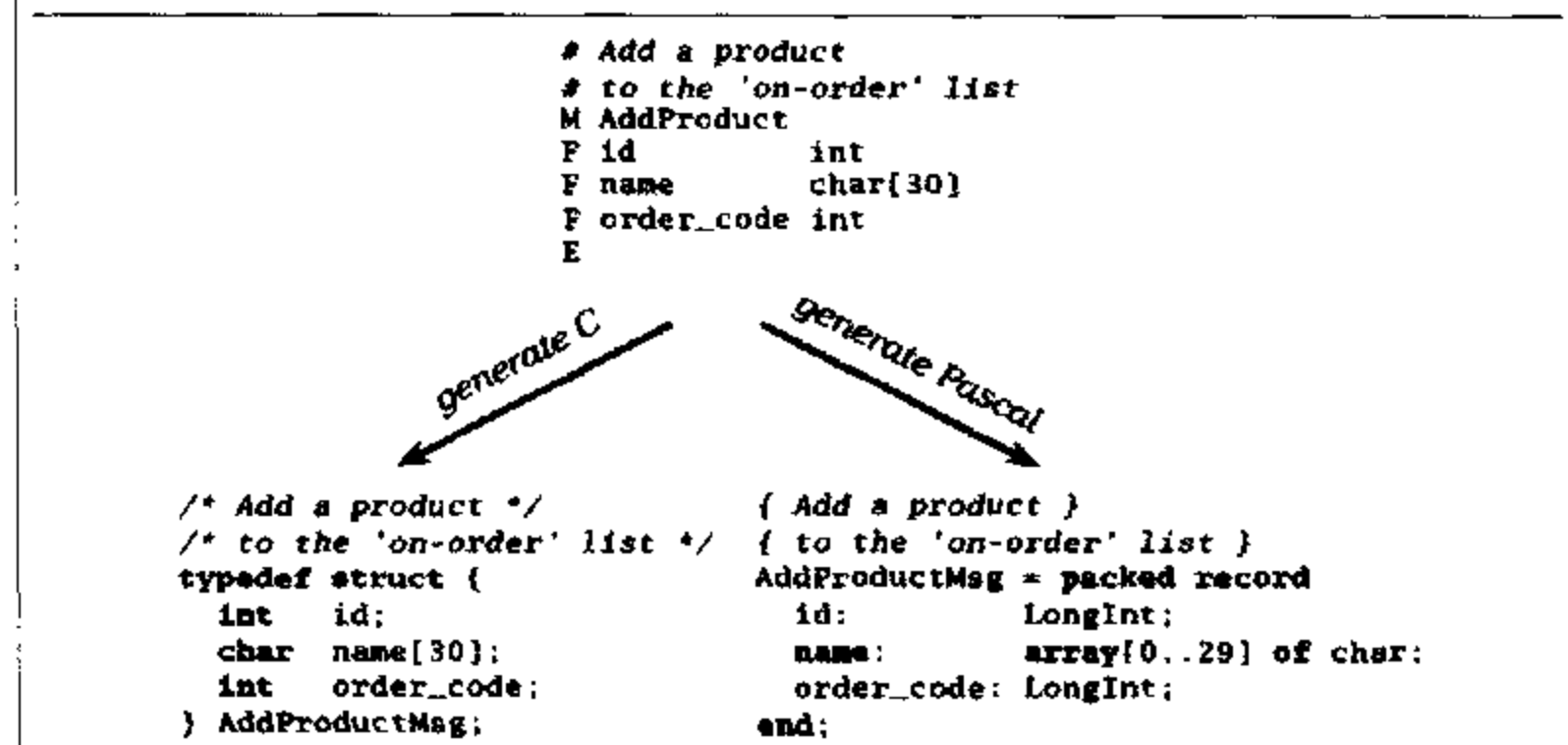
使用代码生成器融合环境的另一个例子发生在不同的编程语言被用于同一个应用时。为了进行通信，每个代码库将需要某些公共信息——例如，数据结构、消息格式、以及字段名。要使用代码生成器，而不是重复这些信息。有时你可以从一种语言的源文件中解析出信息，并将其用于生成第二种语言的代码。但如下一页的图 3.4 所示，用更简单、语言中立的表示形式来表示它，并为两种语言生成代码，常常更简单。再看一看 268 页上练习 13 的解答，里面有怎样把对平板文件表示的解析与代码生成分离开来的例子。

代码生成不一定要很复杂

所有这些关于“主动这个”和“被动那个”的谈论可能会给你留下这样的印象：代码生成器是复杂的东西。它们不一定要很复杂。最复杂的部分通常是负责分析输入文件的解析器。让输入格式保持简单，代码生成器就会变得简单。看一看练习 13 的解答（286 页）：实际的代码生成基本上是 print 语句。

²⁴ 你怎样根据数据库 schema 构建代码呢？有几种途径。如果 schema 是保存在平板文件中的（例如，作为 create table 语句），那么一个相对简单的脚本可以解析它并生成源码。另外，如果你使用某种工具在数据库中直接创建 schema，那么你就应该能够直接从数据库的数据词典中提取你需要的信息。Perl 提供了一些库，能让你访问大多数重要的数据库。

图 3.4 根据语言中立的表示生成代码 在输入文件中，以‘M’开始的行标志着消息定义的开始。‘F’行定义字段，‘E’是消息的结束



代码生成器不一定要生成代码

尽管本节的许多例子给出的是生成程序源码的代码生成器，事情并不是非如此不可。你可以用代码生成器生成几乎任何输出：HTML、XML、纯文本——可能成为你的项目中别处输入的任何文本。

相关内容：

- 重复的危害，26 页
- 纯文本的力量，73 页
- 邪恶的向导，198 页
- 无处不在的自动化，230 页

练习

13. 编写一个代码生成器，读取图 3.4 中的输入文件，以你选择的两种语言生成输出。设法使它容易增加新语言。（解答在 286 页）

第 4 章

注重实效的偏执 Pragmatic Paranoia

提示 30

You Can't Write Perfect Software

你不可能写出完美的软件

这刺痛了你？不应该。把它视为生活的公理，接受它，拥抱它，庆祝它。因为完美的软件不存在。在计算技术简短的历史中，没有一个人曾经写出过一个完美的软件。你也不大可能成为第一个。除非你把这作为事实接受下来，否则你最终会把时间和精力浪费在追逐不可能实现的梦想上。

那么，给定了这个让人压抑的现实，**注重实效的程序员**怎样把它转变为有利条件？这正是这一章的话题。

每个人都知道只有他们自己是地球上的好司机。所有其他的人都等在那里要对他们不利，这些人乱冲停车标志、在车道之间摇来摆去，不作出转向指示、打电话、看报纸，总而言之就是不符合我们的标准。于是我们防卫性地开车。我们在麻烦发生之前小心谨慎，预判意外之事，从不让自己陷入无法解救自己的境地。

编码的相似性相当明显。我们不断地与他人的代码接合——可能不符合我们的高标准的代码——并处理可能有效，也可能无效的输入。所以我们被教导说，要防

卫性地编码。如果有任何疑问，我们就会验证给予我们的所有信息。我们使用断言检测坏数据。我们检查一致性，在数据库的列上施加约束，而且通常对自己感到相当满意。

但**注重实效的程序员**会更进一步。他们连自己也不信任，知道没有人能编写完美的代码，包括自己，所以**注重实效的程序员**针对自己的错误进行防卫性的编码。我们将在“按合约设计（Design by Contract）”中描述第一种防卫措施：客户与供应者必须就权利与责任达成共识。

在“死程序不说谎”中，我们想要确保在找出 bug 的过程中，不会造成任何破坏。所以我们设法经常检查各种事项，并在程序出问题终止程序。

“断言式编程”描述了一种沿途进行检查的轻松方法——编写主动校验你的假定的代码。

与其他任何技术一样，异常如果没有得到适当使用，造成的危害可能比带来的好处更多。我们将在“何时使用异常”中讨论各种相关问题。

随着你的程序变得更为动态，你会发现自己在用系统资源玩杂耍——内存、文件、设备，等等。在“怎样配平资源（How to Balance Resources）”中，我们将提出一些方法，确保你不会让其中任何一个球掉落下来。

不完美的系统、荒谬的时间标度、可笑的工具、还有不可能实现的需求——在这样一个世界上，让我们安全“驾驶”。

当每个人都确实要对你不利时，偏执就是一个好主意。

——Woody Allen

21 按合约设计

没有什么比常识和坦率更让人感到惊讶。

——拉尔夫·沃尔多·爱默生，《散文集》

与计算机系统打交道很困难。与人打交道更困难。但作为一个族类，我们花费在弄清楚人们交往的问题上的时间更长。在过去几千年中我们得出的一些解决办法也可应用于编写软件。确保坦率的最佳方案之一就是合约。

合约既规定你的权利与责任，也规定对方的权利与责任。此外，还有关于任何一方没有遵守合约的后果的约定。

或许你有一份雇用合约，规定了你的工作时数和你必须遵循的行为准则。作为回报，公司付给你薪水和其他津贴。双方都履行其义务，每个人都从中受益。

全世界都——正式地或非正式地——采用这种理念帮助人们交往。我们能否采用同样的概念帮助软件模块进行交互？答案是肯定的。

DBC

Bertrand Meyer[Mey97b]为 Eiffel 语言发展了按合约设计的概念²⁵。这是一种简单而强大的技术，它关注的是用文档记载（并约定）软件模块的权利与责任，以确保程序正确性。什么是正确的程序？不多不少，做它声明要做的事情的程序。用文档记载这样的声明，并进行校验，是按合约设计（简称 DBC）的核心所在。

软件系统中的每一个函数和方法都会做某件事情。在开始做某事之前，例程对世界的状态可能有某种期望，并且也可能有能力陈述系统结束时的状态。Meyer 这样描述这些期望和陈述：

²⁵ 部分地以 Dijkstra、Floyd、Hoare、Wirth 及其他一些人早先的工作为基础。要了解更多关于 Eiffel 自身的信息，参见[URL 10]与[URL 11]。

- **前条件 (precondition)**。为了调用例程，必须为真的条件；例程的需求 在其前条件被违反时，例程决不应被调用。传递好数据是调用者的责任（见 115 页的方框）
- **后条件 (postcondition)**。例程保证会做的事情，例程完成时世界的状态 例程有后条件这一事实意味着它会结束：不允许有无限循环。
- **类不变项 (class invariant)**。类确保从调用者的视角来看，该条件总是为真。在例程的内部处理过程中，不变项不一定会保持，但在例程退出、控制返回到调用者时，不变项必须为真（注意，类不能给出无限制的对参与不变项的任何数据成员的写访问）

让我们来看一个例程的合约，它把数据值插入惟一、有序的列表中。在 iContract（用于 Java 的预处理器，可从[URL 17]获取）中，你可以这样指定：

```
/**
 * @invariant forall Node n in elements() {
 *     n.prev() != null
 *     implies
 *         n.value().compareTo(n.prev().value()) > 0
 * }
 */
public class dbc_list {
    /**
     * @pre contains(aNode) == false
     * @post contains(aNode) == true
     */
    public void insertNode(final Node aNode) {
        // ...
    }
}
```

这里我们所说的是，这个列表中的节点必须以升序排列。当你插入新节点时，它不能是已经存在的，我们还保证，在你插入某个节点后，你将能够找到它。

你用目标编程语言（或许还有某些扩展）编写这些前条件、后条件以及不变项。例如，除了普通的 Java 构造体，iContract 还提供了谓词逻辑操作符——forall、exists、还有 implies。你的断言可以查询方法能够访问的任何对象的状态，但要确保查询没有任何副作用（参见 124 页）。

DBC 与常量参数

后条件常常要使用传入方法的参数来校验正确的行为。但如果允许例程改变传入的参数，你就有可能规避合约。Eiffel 不允许这样的事情发生，但 Java 却允许。这里，我们使用 Java 关键字 `final` 指示我们的意图：参数在方法内不应被改变。这并非十分安全——子类有把参数重新声明为非 `final` 的自由。另外，你可以使用 iContract 语法 `variable@pre` 获取变量在进入方法时的初始值。

这样，例程与任何潜在的调用者之间的合约可解读为：

如果调用者满足了例程的所有前条件，例程应该保证在其完成时，所有后条件和不变项将为真。

如果任何一方没有履行合约的条款，（先前约定的）某种补偿措施就会启用——例如，引发异常或是终止程序。不管发生什么，不要误以为没能履行合约是 bug。它不是某种决不应该发生的事情，这也就是为什么前条件不应被用于完成像用户输入验证这样的任务的原因。

提示 31

Design with Contracts
通过合约进行设计

在“正交性”（34 页）中，我们建议编写“羞怯”的代码。这里，强调的重点是在“懒惰”的代码上：对在开始之前接受的东西要严格，而允诺返回的东西要尽可能少。记住，如果你的合约表明你将接受任何东西，并允诺返回整个世界，那你就有大量代码要写了！

继承和多态是面向对象语言的基石，是合约可以真正闪耀的领域。假定你正在使用继承创建“是一种（is-a-kind-of）”关系，即一个类是另外一个类的“一种”。你或许会想要坚持 Liskov 替换原则（Lis88）：

子类必须要能通过基类的接口使用，而使用者无须知道其区别

换句话说，你想要确保你创建的新子类型确实是基类型的“一种”——它支持同样的方法，这些方法有同样的含义。我们可以通过合约来做到这一点。要让合约自动应用于将来的每个子类，我们只须在基类中规定合约一次。子类可以（可选地）接受范围更广的输入，或是作出更强的保证，但它所接受的和所保证的至少与其父类一样多。

例如，考虑 Java 基类 `java.awt.Component`。你可以把 AWT 或 Swing 中的任何可视组件当作 `Component`，而不用知道实际的子类是按钮、画布、菜单，还是别的什么。每个个别的组件都可以提供额外的、特殊的功能，但它必须至少提供 `Component` 定义的基本能力。但并没有什么能阻止你创建 `Component` 的一个子类型，提供名称正确、但所做事情却不正确的方法。你可以很容易地创建不进行绘制的 `paint` 方法，或是不设置字体的 `setFont` 方法。AWT 没有用于抓住你没有履行合约的事实的合约。

没有合约，编译器所能做的只是确保子类符合特定的方法型构（signature），但如果我们适当设定基类合约，我们现在就能够确保将来任何子类都无法改变我们的方法的含义。例如，你可能想要这样为 `setFont` 建立合约，确保你设置的字体就是你得到的字体：

```
/**
 * @pre f != null
 * @post getFont() == f
 */
public void setFont(final Font f) {
    // ...
}
```

实现 DBC

使用 DBC 的最大好处也许是它迫使需求与保证的问题走到前台来。在设计时简

单地列举输入域的范围是什么、边界条件是什么、例程允诺交付什么——或者，更重要的，它不允诺交付什么——是向着编写更好的软件的一次飞跃。不对这些事项作出陈述，你就回到了靠巧合编程（参见 172 页），那是许多项目开始、结束、失败的地方。

如果语言不在代码中支持 DBC，你也许就只能走这么远了——这并不太坏。毕竟，DBC 是一种设计技术，即使没有自动检查，你也可以把合约作为注释放在代码中，并仍然能够得到非常实际的好处。至少，在遇到麻烦时，用注释表示的合约给了你一个着手的地方。

断言

尽管用文档记载这些假定是一个了不起的开始，让编译器为你检查你的合约，你能够获得多得多的好处。在有些语言中，你可以通过断言（参见断言式编程，122 页）对此进行部分的模拟。为何只是部分的？你不能用断言做 DBC 能做的每一件事情吗？

遗憾的是，答案是“不能”。首先，断言不能沿着继承层次向下遗传。这就意味着，如果你重新定义了某个具有合约的基类方法，实现该合约的断言不会被正确调用（除非你在新代码中手工复制它们）。在退出每个方法之前，你必须记得手工调用类不变项（以及所有的基类不变项）。根本的问题是合约不会自动实施。

还有，不存在内建的“老”值概念。也就是，与存在于方法入口处的值相同的值。如果你使用断言实施合约，你必须给前条件增加代码，保存你想要在后条件中使用的任何信息。把它与 iContract 比较一下，其后条件可以引用“*variable@pre*”；或者与 Eiffel 比较一下，它支持“老表达式”。

最后，runtime 系统和库的设计不支持合约，所以它们的调用不会被检查。这是一个很大的损失，因为大多数问题常常是在你的代码和它使用的库之间的边界上检测到的（更详细的讨论，参见死程序不说谎，120 页）。

语言支持

有内建的 DBC 支持的语言（比如 Eiffel 和 Sather[URL 12]）自动在编译器和 runtime 系统中检查前条件和后条件。在这样的情况下，你能获得最大的好处，因为所有的代码库（还有库函数）必须遵守它们的合约。

但像 C、C++ 和 Java 这样的更流行的语言呢？对于这些语言，有一些预处理器能够处理作为特殊注释嵌入在原始源码中的合约。预处理器会把这些注释展开成检验断言的代码。

对于 C 和 C++，你可以研究一下 Nana[URL 18]。Nana 不处理继承，但它却能以一种新颖的方式，使用调试器在运行时监控断言。

对于 Java，可以使用 iContract[URL 17]。它读取（JavaDoc 形式的）注释，生成新的包含了断言逻辑的源文件。

预处理器没有内建设施那么好。把它们集成进你的项目可能会很杂乱，而且你使用的其他库没有合约。但它们仍然很有助益；当某个问题以这样的方式被发现时——特别是你本来决不会发现的问题——那几乎像是魔术。

DBC 与早崩溃

DBC 相当符合我们关于早崩溃的概念（参见“死程序不说谎”，120 页）。假定你有一个计算平方根的方法（比如在 Eiffel 的 DOUBLE 类中）。它需要一个前条件，把参数域限制为正数。Eiffel 的前条件通过关键字 `require` 声明，后条件通过 `ensure` 声明，所以你可以编写：

```
sqrt: DOUBLE is
    -- Square root routine
    require
        sqrt_arg_must_be_positive: Current >= 0;
    --- ...
    --- calculate square root here
    --- ...
    ensure
        ((Result*Result) - Current).abs <= epsilon*Current.abs;
        -- Result should be within error tolerance
    end;
```

谁负责？

谁负责检查前条件，是调用者，还是被调用的例程？如果作为语言的一部分实现，答案是两者都不是：前条件是在调用者调用例程之后，但在进入例程自身之前，在幕后测试的。因而如果要对参数进行任何显式的检查，就必须由调用者来完成，因为例程自身永远也不会看到违反了其前条件的参数（对于没有内建支持的语言，你需要用检查这些断言的“前言”（preamble）和/或“后文”（postamble）把被调用的例程括起来）

考虑一个程序，它从控制台读取数字，（通过调用 `sqrt`）计算其平方根，并打印结果。`sqrt` 函数有一个前条件——其参数不能为负。如果用户在控制台上输入负数，要由调用代码确保它不会被传给 `sqrt`。该调用代码有许多选择：它可以终止，可以发出警告并读取另外的数，也可以把这个数变成正数，并在 `sqrt` 返回的结果后面附加一个“f”。无论其选择是什么，这都肯定不是 `sqrt` 的问题。

通过在 `sqrt` 例程的前条件中表示平方根函数的参数域，你把保证正确性的负担转交给了调用者——本应如此。随后你可以在知道了其输入会落在有效范围内的前提下，安全地设计 `sqrt` 例程。

如果你用于计算平方根的算法失败了（或不在规定的错误容忍程度之内），你会得到一条错误消息，以及用于告诉你调用链的栈踪迹（stack trace）。

如果你传给 `sqrt` 一个负参数，Eiffel runtime 会打印错误“`sqrt_arg_must_be_positive`”，还有栈踪迹。这比像 Java、C 和 C++ 等语言中的情况要好，在这些语言那里，把负数传给 `sqrt`，返回的是特殊值 NaN（Not a Number）。要等到你随后在程序中试图对 NaN 进行某种运算时，你才会得到让你吃惊的结果。

通过早崩溃、在问题现场找到和诊断问题要容易得多。

不变项的其他用法

到目前为止，我们已经讨论了适用于单个方法的前条件和后条件，以及应用于类中所有方法的不变项，但使用不变项还有其他一些有用的方式。

循环不变项

在复杂的循环上正确设定边界条件可能会很成问题。循环常有香蕉问题（我知道怎样拼写“banana”，但不知道何时停下来——“bananana...”）、篱笆桩错误（不知道该数桩还是该数空）、以及无处不在的“差一个”错误[URL 52]

在这些情况下，不变项可以有帮助：循环不变项是对循环的最终目标的陈述，但又进行了一般化，这样在循环执行之前和每次循环迭代时，它都是有效的。你可以把它视为一种微型合约。经典的例子是找出数组中的最大值的例程：

```
int m = arr[0]; // example assumes arr.length > 0
int i = 1;

// Loop invariant: m = max(arr[0:i-1])
while (i < arr.length) {
    m = Math.max(m, arr[i]);
    i = i + 1;
}
```

（`arr[m:n]`是便捷表示法，意为数组从下标 m 到 n 的部分。）不变项在循环运行之前必须为真，循环的主体必须确保它在循环执行时保持为真。这样我们就知道不变项在循环终止时也保持不变，因而我们的结果是有效的。循环不变项可被显式地编写成断言，但作为设计和文档工具，它们也很有用。

语义不变项

你可以使用语义不变项（semantic invariant）表达不可违反的需求，一种“哲学合约”。

我们曾经编写过一个借记卡交易交换程序。一个主要的需求是借记卡用户的同一笔交易不能被两次记录到账户中。换句话说，不管发生何种方式的失败，结果都应该

是：不处理交易，而不是处理重复的交易

这个简单的法则，直接由需求驱动，被证明非常有助于处理复杂的错误恢复情况，并且可以在许多领域中指导详细的设计和实现

一定不要把固定的需求、不可违反的法则与那些仅仅是政策（policy）的东西混为一谈，后者可能会随着新的管理制度的出台而改变。这就是我们为什么要使用术语“语义不变项”的原因——它必须是事物的确切含义的中心，而不受反复无常的政策支配（后者是更为动态的商业规则的用途所在）

当你发现合格的需求时，确保让它成为你制作的无论什么文档的一个众所周知的部分——无论它是一式三份签署的需求文档中的圆点列表，还是只是每个人都能看到的公共白板上的重要通知。设法清晰、无歧义地陈述它。例如，在借记卡的例子中，我们可以写：

出错时要偏向消费者

这是清楚、简洁、无歧义的陈述，适用于系统的许多不同的区域。它是我们与系统的所有用户之间的合约，是我们对行为的保证。

动态合约与代理

直到现在为止，我们一直把合约作为固定的、不可改变的规范加以谈论。但在自治代理（autonomous agent）的领域中，情况并不一定是这样。按照“自治”的定义，代理有拒绝它们不想接受的请求的自由——“我无法提供那个，但如果你给我这个，那么我可以提供另外的某样东西”

无疑，任何依赖于代理技术的系统对合约协商的依赖都是至关重要的——即使它们是动态生成的。

设想一下，通过足够的“能够互相磋商合约、以实现某个目标”的组件和代理，我们也许就能解决软件生产率危机：让软件为我们解决它。

但如果我们不能手工使用合约，我们也无法自动使用它们。所以下次你设计软件时，也要设计它的合约

相关内容:

- 正交性, 34 页
- 死程序不说谎, 120 页
- 断言式编程, 122 页
- 怎样配平资源, 129 页
- 解耦与得墨忒耳法则, 138 页
- 时间耦合, 150 页
- 靠巧合编程, 172 页
- 易于测试的代码, 189 页
- 注重实效的团队, 224 页

挑战

- 思考这样的问题: 如果 DBC 如此强大, 它为何没有得到更广泛的使用? 制定合约困难吗? 它是否会让你思考你本来想先放在一边的问题? 它迫使你思考吗? 显然, 这是一个危险的工具!

练习

14. 好合约有什么特征? 任何人都可以增加前条件和后条件, 但那是否会给你带来任何好处? 更糟糕的是, 它们实际上带来的坏处是否会大过好处? 对于下面的以及练习 15 和 16 中的例子, 确定所规定的合约是好、是坏、还是很糟糕, 并解释为什么

首先, 让我们看一个 Eiffel 例子。我们有一个用于把 STRING 添加到双向链接的循环链表中的例程 (别忘了前条件用 `require` 标注, 后条件用 `ensure` 标注)

(解答在 288 页)

```
-- Add an item to a doubly linked list,
-- and return the newly created NODE.
add_item (item : STRING) : NODE is
  require
    item /= Void -- '/=' is 'not equal'.
  deferred -- Abstract base class.
  ensure
    result.next.previous = result -- Check the newly
    result.previous.next = result -- added node's links.
    find_item(item) = result      -- Should find it.
  End
```

15. 下面，让我们试一试一个 Java 的例子——与练习 14 中的例子有点类似。

`insertNumber` 把整数插入有序列表中。前条件和后条件的标注方式与 `iContract`（参见[URL 17]）一样。（解答在 288 页）

```
private int data[];
/**
 * @post data[index-1] < data[index] &&
 *       data[index] == aValue
 */
public Node insertNumber (final int aValue)
{
    int index = findPlaceToInsert(aValue);
    ...
}
```

16. 下面的代码段来自 Java 的栈类。这是好合约吗？（解答在 289 页）

```
/**
 * @pre anItem != null // Require real data
 * @post pop() == anItem // Verify that it's
 *       // on the stack
 */
public void push(final String anItem)
```

17. DBC 的经典例子（如练习 14-16 中的例子）给出的是某种 ADT（Abstract Data Type）的实现——栈或队列就是典型的例子，但并没有多少人真的会编写这种低级的类。

所以，这个练习的题目是，设计一个厨用搅拌机接口。它最终将是一个基于 Web、适用于 Internet、CORBA 化的搅拌机，但现在我们只需要一个接口来控制它。它有十挡速率设置（0 表示关机）。你不能在它空的时候进行操作，而且你只能一挡一挡地改变速率（也就是说，可以从 0 到 1，从 1 到 2，但不能从 0 到 2）。

下面是各个方法。增加适当的前条件、后条件和不变项（解答在 289 页）

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

18. 在 0, 5, 10, 15, ..., 100 序列中有多少个数？（解答在 290 页）

22 死程序不说谎

你是否注意到，有时别人在你自己意识到之前就能觉察到你的事情出了问题。别人的代码也是一样。如果我们的某个程序开始出错，有时库例程会最先抓住它。一个“迷途的”指针也许已经致使我们用无意义的内容覆写了某个文件句柄。对 `read` 的下次调用将会抓住它。或许缓冲区越界已经把我们要用于检测分配多少内存的计数器变成了垃圾。也许我们对 `malloc` 的调用将会失败。数百万条之前的某个逻辑错误意味着某个 `case` 语句的选择开关不再是预期的 1、2 或 3。我们将会命中 `default` 情况（这是为什么每个 `case/switch` 语句都需要有 `default` 子句的原因之一——我们想要知道何时发生了“不可能”的事情）。

我们很容易掉进“它不可能发生”这样一种心理状态。我们中的大多数人编写的代码都不检查文件是否能成功关闭，或者某个跟踪语句是否已按照我们的预期写出。而如果所有的事情都能如我们所愿，我们很可能就不需要那么做——这些代码在任何正常的条件都不会失败。但我们是在防卫性地编程，我们在程序的其他部分中查找破坏堆栈的“淘气指针”，我们在检查确实加载了共享库的正确版本。

所有的错误都能为你提供信息。你可以让自己相信错误不可能发生，并选择忽略它。但与此相反，**注重实效的程序员**告诉自己，如果有一个错误，就说明非常、非常糟糕的事情已经发生了。

提示 32

Crash Early

早崩溃

要崩溃，不要破坏（trash）

尽早检测问题的好处之一是你更早崩溃。而有许多时候，让你的程序崩溃是

你的最佳选择。其他的办法可以是继续执行、把坏数据写到某个极其重要的数据库或是命令洗衣机进入其第二十次连续的转动周期。

Java 语言和库已经采用了这一哲学。当意料之外的某件事情在 runtime 系统中发生时，它会抛出 `RuntimeException`。如果没有被捕捉，这个异常就会渗透到程序的顶部，致使其中止，并显示栈踪迹。

你可以在别的语言中做相同的事情。如果没有异常机制，或是你的库不抛出异常，那么就确保你自己对错误进行了处理。在 C 语言中，对于这一目的，宏可能非常有用：

```
#define CHECK(LINE, EXPECTED) \
{ int rc = LINE; \
  if (rc != EXPECTED) \
    ut_abort(__FILE__, __LINE__, #LINE, rc, EXPECTED); }

void ut_abort(char *file, int ln, char *line, int rc, int exp) {
    fprintf(stderr, "%s line %d\n'%s': expected %d, got %d\n",
            file, ln, line, exp, rc);
    exit(1);
}
```

然后你可以这样包装决不应该失败的调用：

```
CHECK(stat("/tmp", &stat_buff), 0);
```

如果它失败了，你就会得到写到 `stderr` 的消息：

```
source.c line 19
'stat("/tmp", &stat_buff)': expected 0, got -1
```

显然，有时简单地退出运行中的程序并不合适。你申请的资源可能没有释放，或者你可能要写出日志消息，清理打开的事务，或与其他进程交互。我们在“何时使用异常”（125 页）中讨论的技术在此能对你有帮助。但是，基本的原则是一样的——当你的代码发现，某件被认为不可能发生的事情已经发生时，你的程序就不再有存活能力。从此时开始，它所做的任何事情都会变得可疑，所以要尽快终止它。死程序带来的危害通常比有疾患的程序要小得多。

相关内容：

- 按合约设计，109 页
- 何时使用异常，125 页

23 断言式编程

在自责中有一种满足感。当我们责备自己时，会觉得再没人有权责备我们。

——奥斯卡·王尔德：《多里安·格雷的画像》

每一个程序员似乎都必须在其职业生涯的早期记住一段曼特罗（mantra）。它是计算技术的基本原则，是我们学着应用于需求、设计、代码、注释——也就是我们所做的每一件事情——的核心信仰。那就是：

这决不会发生……

“这些代码不会被用上 30 年，所以用两位数字表示日期没问题。”“这个应用决不会在国外使用，那么为什么要使其国际化？”“count 不可能为负。”“这个 printf 不可能失败。”

我们不要这样自我欺骗，特别是在编码时。

提示 33

If It Can't Happen, Use Assertions to Ensure That It Won't

如果它不可能发生，用断言确保它不会发生

无论何时你发现自己在思考“但那当然不可能发生”，增加代码检查它。最容易的办法是使用断言。在大多数 C 和 C++ 实现中，你都能找到某种形式的检查布尔条件的 `assert` 或 `_assert` 宏。这些宏是无价的财富。如果传入你的过程的指针决不应该是 `NULL`，那么就检查它：

```
void writeString(char *string) {
    assert(string != NULL);
    ...
}
```

对于算法的操作，断言也是有用的检查。也许你编写了一个聪明的排序算法。检查它是否能工作：

```
for (int i = 0; i < num_entries-1; i++) {
    assert(sorted[i] <= sorted[i+1]);
}
```

当然，传给断言的条件不应该有副作用（参见 124 页的方框）。还要记住断言可能

会在编译时被关闭——决不要把必须执行的代码放在 `assert` 中

不要用断言代替真正的错误处理 断言检查的是决不应该发生的事情：你不会想编写这样的代码：

```
printf("Enter 'Y' or 'N': ");
ch = getchar();
assert((ch == 'Y') || (ch == 'N'));    /* bad idea! */
```

而且，提供给你的 `assert` 宏会在断言失败时调用 `exit`，并不意味着你编写的版本就应该这么做。如果你需要释放资源，就让断言失败生成异常、`longjump` 到某个退出点、或是调用错误处理器。要确保你在终止前的几毫秒内执行的代码不依赖最初触发断言失败的信息。

让断言开着

有一个由编写编译器和语言环境的人传播的、关于断言的常见误解。就是像这样的说法：

断言给代码增加了一些开销，因为它们检查的是决不应该发生的事情，所以只会由代码中的 `bug` 触发。一旦代码经过了测试并发布出去，它们就不再需要存在，应该被关闭，以使代码运行得更快。断言是一种调试设施。

这里有两个明显错误的假定。首先，他们假定测试能找到所有的 `bug`。现实的情况是，对于任何复杂的程序，你甚至不大可能测试你的代码执行路径的排列数的极小一部分（参见“无情的测试”，245 页）。其次，乐观主义者忘记了你的程序运行在一个危险的世界里。在测试过程中，老鼠可能不会噬咬通信电缆、某个玩游戏的人不会耗尽内存、日志文件不会塞满硬盘。这些事情可能会在你的程序运行在实际工作环境中时发生。你的第一条防线是检查任何可能的错误，第二条防线是使用断言设法检测你疏漏的错误。

在你把程序交付使用时关闭断言就像是因为你曾经成功过，就不用保护网去走钢丝。那样做有极大的价值，但却难以获得人身保险。

即使你确实有性能问题，也只关闭那些真的有很大影响的断言。上面的排序例子

断言与副作用

如果我们增加的错误检测代码实际上却制造了新的错误，那是一件让人尴尬的事情。如果对条件的计算有副作用，这样的事情可能会在使用断言时发生。例如，在 Java 中，像下面这样编写代码，不是个好主意：

```
while (iter.hasMoreElements () {
    Test.ASSERT(iter.nextElement() != null);
    Object obj = iter.nextElement();
    // ....
}
```

ASSERT 中的 `nextElement()` 调用有副作用：它会让迭代器越过正在读取的元素，这样循环就会只处理集合中的一半元素。这样编写代码会更好：

```
while (iter.hasMoreElements()) {
    Object obj = iter.nextElement();
    Test.ASSERT(obj != null);
    // ....
}
```

这个问题是一种“海森堡虫子”（Heisenbug）——调试改变了被调试系统的行为（参见[URL 52]）。

也许是你的应用的关键部分，也许需要很快才行。增加检查意味着又一次通过数据，这可能让人不能接受。让那个检查成为可选的²⁶，但让其余的留下来。

相关部分：

- 调试，90 页
- 按合约设计，109 页
- 怎样配平资源，129 页
- 靠巧合编程，172 页

²⁶ 在基于 C 的语言中，你可以使用预处理器或 if 语句让断言成为可选的。如果设置（或没有设置）某个编译时标志，许多实现会关闭 `assert` 宏的代码生成。另外，你可以把代码放在具有常量条件的 if 语句中，许多编译器（包括大多数常见 Java 系统）都会把它优化掉。

练习

19. 一次快速的真实性检查 下面这些“不可能”的事情中，那些可能发生？（解答在 290 页）

1. 一个月少于 28 天
2. `stat(".", &sb) == -1`（也就是，无法访问当前目录）
3. 在 C++ 里：`a = 2; b = 3; if (a + b != 5) exit(1);`
4. 内角和不等下 180° 的三角形
5. 没有 60 秒的一分钟
6. 在 Java 中：`(a + 1) <= a`

20. 为 Java 开发一个简单的断言检查类。（解答在 291 页）

24 何时使用异常

在“死程序不说谎”（120 页）中，我们提出，检查每一个可能的错误——特别是意料之外的错误——是一种良好的实践。但是，在实践中这可能会把我们引向相当丑陋的代码；你的程序的正常逻辑最后可能会被错误处理完全遮蔽，如果你赞成“例程必须有单个 `return` 语句”的编程学派（我们不赞成），情况就更是如此。我们见过看上去像这样的代码：

```
retcode = OK;
if (socket.read(name) != OK) {
    retcode = BAD_READ;
}
else {
    processName(name);
    if (socket.read(address) != OK) {
        retcode = BAD_READ;
    }
    else {
        processAddress(address);
        if (socket.read(telNo) != OK) {
            retcode = BAD_READ;
        }
        else {
            // etc, etc...
        }
    }
}
return retcode;
```

幸运的是，如果编程语言支持异常，你可以通过更为简洁的方式重写这段代码：

```
retcode = OK;

try {
    socket.read(name);
    process(name);

    socket.read(address);
    processAddress(address);

    socket.read(telNo);
    // etc, et...
}
catch (IOException e) {
    retcode = BAD_READ;
    Logger.log("Error reading individual: " + e.getMessage());
}
return retcode;
```

现在正常的控制流很清晰，所有的错误处理都移到了一处。

什么是异常情况

关于异常的问题之一是知道何时使用它们。我们相信，异常很少应作为程序的正常流程的一部分使用；异常应保留给意外事件。假定某个未被抓住的异常会终止你的程序，问问你自己：“如果我移走所有的异常处理器，这些代码是否仍然能运行？”如果答案是“否”，那么异常也许就正在被用在非异常的情形中。

例如，如果你的代码试图打开一个文件进行读取，而该文件并不存在，应该引发异常吗？

我们的回答是：“这取决于实际情况。”如果文件应该在那里，那么引发异常就有正当理由。某件意外之事发生了——你期望其存在的文件好像消失了。另一方面，如果你不清楚该文件是否应该存在，那么你找不到它看来就不是异常情况，错误返回就是合适的。

让我们看一看第一种情况的一个例子。下面的代码打开文件/etc/passwd，这个文件在所有的 UNIX 系统上都应该存在。如果它失败了，它会把 FileNotFoundException 传给它的调用者。

```
public void open_passwd() throws FileNotFoundException {
    // This may throw FileNotFoundException...
    ipstream = new FileInputStream("/etc/passwd");
    // ...
}
```

但是，第二种情况可能涉及打开用户在命令行上指定的文件。这里引发异常没有正当理由，代码看起来也不同：

```
public boolean open_user_file(String name)
    throws FileNotFoundException {
    File f = new File(name);
    if (!f.exists()) {
        return false;
    }
    InputStream = new FileInputStream(f);
    return true;
}
```

注意 `FileInputStream` 调用仍有可能生成异常，这个例程会把它传递出去。但是，这个异常只在真正异常的情形下才生成；只是试图打开不存在的文件将生成传统的错误返回。

提示 34

Use Exceptions for Exceptional Problems

将异常用于异常的问题

我们为何要提出这种使用异常的途径？嗯，异常表示即时的、非局部的控制转移——这是一种级联的（cascading）goto。那些把异常用作其正常处理的一部分的程序，将遭受到经典的意大利面条式代码的所有可读性和可维护性问题的折磨。这些程序破坏了封装：通过异常处理，例程和它们的调用者被更紧密地耦合在一起。

错误处理器是另一种选择

错误处理器是检测到错误时调用的例程。你可以登记一个例程处理特定范畴的错误。处理器会在其中一种错误发生时被调用。

有时你可能想要使用错误处理器，或者用于替代异常，或者与异常一起使用。显然，如果你使用像 C 这样不支持异常的语言，这是你的很少几个选择之一（参见下一

页的“挑战”)。但是,有时错误处理器甚至也可用于拥有良好的内建异常处理方案的语言(比如Java)。

考虑一个客户-服务器应用的实现,它使用了Java的Remote Method Invocation (RMI)设施。因为RMI的实现方式,每个对远地例程的调用都必须准备处理RemoteException。增加代码处理这些异常可能会变得让人厌烦,并且意味着我们难以编写既能与本地例程、也能与远地例程一起工作的代码。一种绕开这一问题的可能方法是把你的远地对象包装在非远地的类中。这个类随即实现一个错误处理器接口,允许客户代码登记一个在检测到远地异常时调用的例程。

相关内容:

- 死程序不说谎, 120 页

挑战

- 不支持异常的语言常常拥有一些其他的非局部控制转移机制(例如,C拥有longjmp/setjmp)。考虑一下怎样使用这些设施实现某种仿造的异常机制。其好处和危险是什么?你需要采取什么特殊步骤确保资源不被遗弃?在你编写的所有C代码中使用这种解决方案有意义吗?

练习

21. 在设计一个新的容器类时,你确定可能有以下错误情况: (解答在 292 页)

- (1) add 例程中的新元素没有内存可用
- (2) 在 fetch 例程中找不到所请求的数据项
- (3) 传给 add 例程的是 null 指针

应怎样处理每种情况?应该生成错误、引发异常、还是忽略该情况?

25 怎样配平资源

“我把你带进这个世界，”我的父亲会说：“我也可以把你赶出去。那没有我影响。我要再造另一个你。”

——Bill Cosby, *Fatherhood*

只要在编程，我们都要管理资源：内存、事务、线程、文件、定时器——所有数量有限的事物。大多数时候，资源使用遵循一种可预测的模式：你分配资源，使用它，然后解除其分配。

但是，对于资源分配和解除分配的处理，许多开发者没有始终如一的计划。所以让我们提出一个简单的提示：

提示 35

Finish What You Start

要有始有终

在大多数情况下这条提示都很容易应用。它只是意味着，分配某项资源的例程或对象应该负责解除该资源的分配。让我们通过一个糟糕的代码例子来看看该提示的应用方式——这是一个打开文件、从中读取消费者信息、更新某个字段、然后写回结果的应用。我们除去了其中的错误处理代码，以让例子更清晰：

```
void readCustomer(const char *fName, Customer *cRec) {
    cFile = fopen(fName, "r+");
    fread(cRec, sizeof(*cRec), 1, cFile);
}

void writeCustomer(Customer *cRec) {
    rewind(cFile);
    fwrite(cRec, sizeof(*cRec), 1, cFile);
    fclose(cFile);
}

void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fName, &cRec);
    cRec.balance = newBalance;
    writeCustomer(&cRec);
}
```

初看上去，例程 `updateCustomer` 相当好。它似乎实现了我们所需的逻辑——

读取记录，更新余额，写回记录。但是，这样的整洁掩盖了一个重大的问题。例程 `readCustomer` 和 `writeCustomer` 紧密地耦合在一起²⁷——它们共享全局变量 `cFile`。`readCustomer` 打开文件，并把文件指针存储在 `cFile` 中，而 `writeCustomer` 使用所存储的指针在其结束时关闭文件。这个全局变量甚至没有出现在 `updateCustomer` 例程中。

这为什么不好？让我们考虑一下，不走运的维护程序员被告知规范发生了变化——余额只应在新的值不为负时更新。她进入源码，改动 `updateCustomer`：

```
void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fName, &cRec);
    if (newBalance >= 0.0) {
        cRec.balance = newBalance;
        writeCustomer(&cRec);
    }
}
```

在测试时一切似乎都很好。但是，当代码投入实际工作，若干小时后它就崩溃了，抱怨说打开的文件太多。因为 `writeCustomer` 在有些情形下不会被调用，文件也就不会被关闭。

这个问题的一个非常糟糕的解决方案是在 `updateCustomer` 中对该特殊情况进行处理：

```
void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fName, &cRec);
    if (newBalance >= 0.0) {
        cRec.balance = newBalance;
        writeCustomer(&cRec);
    }
    else
        fclose(cFile);
}
```

这可以修正问题——不管新的余额是多少，文件现在都会被关闭——但这样的修

²⁷ 关于耦合代码的危险的讨论，参见“解耦与得墨忒耳法则”（138页）。

正意味着三个例程通过全局的 `cFile` 耦合在一起。我们在掉进陷阱，如果我们继续沿着这一方向前进，事情就会开始迅速变糟。

要有始有终这一提示告诉我们，分配资源的例程也应该释放它。通过稍稍重构代码，我们可以在此应用该提示：

```
void readCustomer(FILE *cFile, Customer *cRec) {
    fread(cRec, sizeof(*cRec), 1, cFile);
}

void writeCustomer(FILE *cFile, Customer *cRec) {
    rewind(cFile);
    fwrite(cRec, sizeof(*cRec), 1, cFile);
}

void updateCustomer(const char *fName, double newBalance) {
    FILE *cFile;
    Customer cRec;

    cFile = fopen(fName, "r+");           // >--
    readCustomer(cFile, &cRec);           //      /
    if (newBalance >= 0.0) {               //      /
        cRec.balance = newBalance;        //      /
        writeCustomer(cFile, &cRec);      //      /
    }                                       //      /
    fclose(cFile);                         // <--
}
```

现在 `updateCustomer` 例程承担了关于该文件的所有责任。它打开文件并（有始有终地）在退出前关闭它。例程配平了对文件的使用：打开和关闭在同一个地方，而且显然每一次打开都有对应的关闭。重构还移除了丑陋的全局变量。

嵌套的分配

对于一次需要不只一个资源的例程，可以对资源分配的基本模式进行扩展。有两个另外的建议：

1. 以与资源分配的次序相反的次序解除资源的分配。这样，如果一个资源含有对另一个资源的引用，你就不会造成资源被遗弃。
2. 在代码的不同地方分配同一组资源时，总是以相同的次序分配它们。这将降低发生死锁的可能性。（如果进程 A 申请了 `resource1`，并正要申请 `resource2`，而进程 B 申请了 `resource2`，并试图获得 `resource1`，这两个进程就会永远等待下去。）

不管我们在使用的是何种资源——事务、内存、文件、线程、窗口——基本的模式

都适用：

无论是谁分配的资源，它都应该负责解除该资源的分配。但是，在有些语言中，我们可以进一步发展这个概念。

对象与异常

分配与解除分配的对称让人想起类的构造器与析构器。类代表某个资源，构造器给予你该资源类型的特定对象，而析构器将其从你的作用域中移除。

如果你是在用面向对象语言编程，你可能会发现把资源封装在类中很有用。每次你需要特定的资源类型时，你就实例化这个类的一个对象。当对象出作用域或是被垃圾收集器回收时，对象的析构器就会解除所包装资源的分配。

配平与异常

支持异常的语言可能会使解除资源的分配很棘手。如果有异常被抛出，你怎样保证在发生异常之前分配的所有资源都得到清理？答案在一定程度上取决于语言。

在 C++ 异常机制下配平资源

C++ 支持 `try...catch` 异常机制。遗憾的是，这意味着在退出某个捕捉异常，并随即将其重新抛出的例程时，总是至少有两条可能的路径：

```
void doSomething(void) {  
    Node *n = new Node;  
  
    try {  
        // do something  
    }  
    catch (...) {  
        delete n;  
        throw;  
    }  
    delete n;  
}
```

注意我们创建的节点是在两个地方释放的——一次是在例程正常的退出路径上，一次是在异常处理器中。这显然违反了 *DRY* 原则，可能会发生维护问题。

但是，我们可以对 C++ 的语义加以利用。局部对象在从包含它们的块中退出时会被自动销毁。这给了我们一些选择。如果情况允许，我们可以把 “n” 从指针改变为栈上实际的 Node 对象：

```
void doSomething1(void) {
    Node n;
    try {
        // do something
    }
    catch (...) {
        throw;
    }
}
```

在这里，不管是否抛出异常，我们都依靠 C++ 自动处理 Node 对象的析构。

如果不可能不使用指针，可以通过在另一个类中包装资源（在这个例子中，资源是一个 Node 指针）获得同样的效果。

```
// Wrapper class for Node resources
class NodeResource {
    Node *n;

public:
    NodeResource() { n = new Node; }
    ~NodeResource() { delete n; }

    Node *operator->() { return n; }
};

void doSomething2(void) {
    NodeResource n;

    try {
        // do something
    }
    catch (...) {
        throw;
    }
}
```

现在包装类 NodeResource 确保了在其对象被销毁时，相应的节点也会被销毁。为了方便起见，包装提供了解引用操作符 `->`，这样它的使用者可以直接访问所包含的 Node 对象中的字段。

因为这一技术是如此有用，标准 C++ 库提供了模板类 `auto_ptr`，能自动包装动态分配的对象

```
void doSomething3(void) {
    auto_ptr<Node> p (new Node);
    // Access the Node as p->...
    // Node automatically deleted at end
}
```

在 Java 中配平资源

与 C++ 不同，Java 实现的是自动对象析构的一种“懒惰”形式——未被引用的对象被认为是垃圾收集的候选者，如果垃圾收集器回收它们，它们的 `finalize` 方法就会被调用。尽管这为开发者提供了便利，他们不再须要为大多数内存泄漏承受指责，但同时也使得实现 C++ 方式的资源清理变得很困难。幸运的是，Java 语言的设计者考虑周详地增加了一种语言特性进行补偿：`finally` 子句。当 `try` 块含有 `finally` 子句时，如果 `try` 块中有任何语句被执行，该子句中的代码就保证会被执行。是否有异常抛出没有影响（即或 `try` 块中的代码执行了 `return` 语句）——`finally` 子句中的代码都将会运行。这意味着我们可以通过这样的代码配平我们的资源使用：

```
public void doSomething() throws IOException {

    File tmpFile = new File(tmpFileName);
    FileWriter tmp = new FileWriter(tmpFile);

    try {
        // do some work
    }
    finally {
        tmpFile.delete();
    }
}
```

该例程使用了一个临时文件，不管例程怎样退出，我们都要删除该文件。`finally` 块使得我们能够简洁地表达这一意图。

当你无法配平资源时

有时基本的资源分配模式并不合适。这通常会出现在使用动态数据结构的程序中。

一个例程将分配一块内存区，并把它链接进某个更大的数据结构中，这块内存可能会在那里呆上一段时间

这里的诀窍是为内存分配设立一个语义不变项。你须要决定谁为某个聚集数据结构(aggregate data structure)中的数据负责。当你解除顶层结构的分配时会发生什么？你有三个主要选择：

1. 顶层结构还负责释放它包含的任何子结构。这些结构随即递归地删除它们包含的数据，等等
2. 只是解除顶层结构的分配。它指向的（没有在别处引用的）任何结构都会被遗弃
3. 如果顶层结构含有任何子结构，它就拒绝解除自身的分配

这里的选择取决于每个数据结构自身的情形。但是，对于每个结构，你都须明确做出选择，并始终如一地实现你的选择。在像 C 这样的过程语言中实现其中的任何选择都可能会成问题：数据结构自身不是主动的。在这样的情形下，我们的偏好是为每个重要结构编写一个模块，为该结构提供分配和解除分配设施（这个模块也可以提供像调试打印、序列化、解序列化和遍历挂钩这样的设施）。

最后，如果追踪资源很棘手，你可以通过在动态分配的对象上实现一种引用计数方案，编写自己有限的自动垃圾回收机制。*More Effective C++*[Mey96]一书专设了一节讨论这一话题。

检查配平

因为**注重实效的程序员**谁也不信任，包括我们自己，所以我们觉得，构建代码、对资源确实得到了适当释放进行实际检查，这总是一个好主意。对于大多数应用，这通常意味着为每种资源类型编写包装，并使用这些包装追踪所有的分配和解除分配。在你的代码中的特定地方，程序逻辑将要求资源处在特定的状态中：使用包装对此进行检查。

例如，一个长期运行的、对请求进行服务的程序，很可能会在其主处理循环的顶部的某个地方等待下一个请求到达。这是确定自从上次循环执行以来，资源使用未曾增长的好地方。

在一个更低、但用处并非更少的层面上，你可以投资购买能检查运行中的程序的内存泄漏情况（及其他情况）的工具。Purify（www.rational.com）和 Insure++（www.parasoft.com）是两种流行的选择。

相关内容：

- 按合约设计，109 页
- 断言式编程，122 页
- 解耦与得墨忒耳法则，138 页

挑战

- 尽管没有什么途径能够确保你总是释放资源，某些设计技术，如果能够始终如一地加以应用，将能对你有所帮助。在上文中我们讨论了为重要数据结构设立语义不变项可以怎样引导内存解除分配决策。考虑一下，“按合约设计”（109 页）可以怎样帮助你提炼这个想法。

练习

22. 有些 C 和 C++ 开发者故意在解除了某个指针引用的内存的分配之后，把该指针设为 NULL。这为什么是个好主意？（解答在 292 页）
23. 有些 Java 开发者故意在使用完某个对象之后，把该对象变量设为 NULL，这为什么是个好主意？（解答在 292 页）

第 5 章

弯曲，或折断 Bend, or Break

生活不会停步不前

我们编写的代码也不会。为了让我们赶上今天近乎疯狂的变化步伐，我们须要尽一切努力编写尽可能宽松——灵活——的代码。否则，我们可能就会发现我们的代码很快就变得过时，或是太脆弱，以至于难以修理，并且最终可能会在向着未来的疯狂突进中掉队。

在“可撤销性”（44 页）中，我们谈到不可撤销的决策的危险。在本章中，我们将告诉你怎样做出可撤销的决策，以使你的代码在面对不确定的世界时保持灵活性和可适应性。

首先我们需要看一看耦合——代码模块间的依赖关系。在“解耦与得墨忒耳法则”中，我们将说明怎样让分离的概念保持分离，并降低耦合程度。

保持灵活的一种好办法是少写代码。改动代码会使你引入新 bug 的可能性增大。“元程序设计”将解释怎样把各种细节完全移出代码，那样就可以更安全、更容易地改动它们。

在“时间耦合”中，我们将看一看时间与耦合相关的两个方面。你是否依赖于先于“嗒”的“嘀”？如果你想要保持灵活，就不要这样做。

创建灵活代码的一个关键概念是数据模型（model）与该模型的视图（view 即表现）的分离。我们将在“它只是视图”中解除模型与视图的耦合。

最后，有一种技术可用于更进一步解除模块的耦合：提供一个“聚会地点”，各模块可以在那里匿名和异步地交换数据。这是黑板这一节的话题。

装备了这些技术，你就可以编写会“摇滚”的代码了。

26 解耦与得墨忒耳法则

好篱笆促成好邻居。

——罗伯特·弗罗斯特，《补墙》

在“正交性”（34页）和“按合约设计”（109页）中，我们提出，编写“羞怯”的代码是有益的。但“羞怯”的工作方式有两种：不向别人暴露你自己，不与太多人打交道。

间谍、反政府人员、革命者以及类似的人常常会组成小组，称为“cell”（最小组织单位）。尽管每个最小组织单位内部的人员可能相互认识，但他们却不认识其他最小组织单位的人。如果某个最小组织单位被发现了，再多的麻醉药也无法使该最小组织单位外部的人的姓名泄漏。切断最小组织单位之间的交往能保护每一个人。

我们觉得，这也是适用于编码的好原则。把你的代码组织成最小组织单位（模块），并限制它们之间的交互。如果随后出于折中必须替换某个模块，其他模块仍能够继续工作。

使耦合减至最少

让模块相互了解有什么问题？原则上没有——我们无需像间谍或反政府人员那样偏执。但是，你确实需要注意你与多少其他模块交互，而且，更重要的是，你是怎样开始与它们交互的。

假定你在改建你的房子，或是从头修建一所房子。典型的安排涉及到找一位“总承包人”，你雇用承包人来完成工作，但承包人可能会、也可能不会亲自进行建造：他

可能会把工作分包给好几个子承包人。但作为客户，你不用直接与这些子承包人打交道——总承包人会替你承担那些让人头疼的事情。

我们想要在软件中遵循同样的模型。当我们要求某个对象完成特定服务时，我们想要它替我们完成该服务。我们不希望这个对象给我们一个第三方对象，我们必须对其加以处理才能获得所需服务。

例如，假定你在编写一个类，生成科学记录仪数据图。你的数据记录仪分散在世界各地；每个记录仪对象都含有一个地点对象，给出其位置及时区。你想要让你的用户选择记录仪，绘制其数据，并标上正确的时区。你可以编写：

```
public void plotDate(Date aDate, Selection aSelection) {
    TimeZone tz =
        aSelection.getRecorder().getLocation().getTimeZone();
    ...
}
```

但现在绘制例程不必要地与三个类耦合在一起——Selection、Recorder 以及 Location。这种编码风格极大地增加了我们的类所依赖的类的数目。这为何是一件坏事？因为它增加了系统别的地方的一个无关改动影响你的代码的风险。例如，如果 Fred 对 Location 做出改动，使它不再直接包含 TimeZone，你也必须改动你的代码。

应该直接要求提供你所需的东西，而不是自行“挖通”调用层次：

```
public void plotDate(Date aDate, TimeZone aTz) {
    ...
}
plotDate(someDate, someSelection.getTimeZone());
```

我们给 Selection 增加了一个方法，让其替我们获取时区：绘制例程不关心时区是直接来自 Recorder，还是来自 Recorder 中包含的某个对象，或者 Selection 是否合成了一个完全不同的时区。选择例程依次又可能只需请求记录仪给出其时区，让记录仪从其包含的 Location 对象那里获取时区。

对象间直接的横贯关系有可能很快带来依赖关系的组合爆炸²⁸。你可以通过若干方式看到这一现象的症状：

1. 这样的 C 或 C++ 大型项目：用于链接某个单元测试的命令比测试程序自身还要长。
2. 对某个模块的“简单”改动会传遍系统中的一些无关模块。
3. 开发者害怕改动代码，因为他们不清楚哪些代码可能会受影响。

有许多不必要的依赖关系的系统非常难以维护（而且很昂贵），往往高度地不稳定。为了使依赖关系保持最少，我们将使用得墨忒耳法则设计我们的方法和函数。

函数的得墨忒耳法则

函数的得墨忒耳法则[LH89]试图使任何给定程序中的模块之间的耦合减至最少。它设法阻止你为了获得对第三个对象的方法的访问而进入某个对象。下一页的图 5.1 对该法则做了总结。

通过编写尽可能遵守得墨忒耳法则的“羞怯”代码，我们可以实现我们的目标：

提示 36

Minimize Coupling Between Modules

使模块之间的耦合减至最少

这真的有关系吗

尽管在理论上听起来很好，遵循得墨忒耳法则真的有助于创建可维护性更好的代码？

一些研究表明[BBM96]，在 C++ 中，与具有较小响应集（response set）的类相比，具有较大响应集（response set）的类更容易出错（响应集的定义是：类的各个方法直接调用的函数的

²⁸ 如果 n 个对象全都互相了解，那么对一个对象的改动就可能对其他 $n-1$ 个对象都需要改动。

图 5.1 函数的得墨忒耳法则

```

class Demeter {
private:
    A *a;
    int func();
public:
    // ...
    void example(B& b);

void Demeter::example(B& b) {
    C c;
    int i = func(); ←---- 它自身

    b.invert(); ←----- 传入该方法的任何参数

    a = new A();
    a->setActive(); ←---- 它创建的任何对象

    c.print(); ←----- 任何直接持有的组件对象
}

```

函数的得墨忒耳法则规定，某个对象的任何方法都应该只调用属于以下情形的方法：

数目)。

因为遵循得墨忒耳法则缩小了调用类（calling class）中的响应集的规模，结果以这种方式设计的类的错误也往往更少（关于得墨忒耳项目的更多论文及信息，参见 [URL56]）

使用得墨忒耳法则将使你的代码适应性更好、更健壮，但也有代价：作为“总承包人”，你的模块必须直接委托并管理全部子承包人，而不牵涉你的模块的客户。在实践中，这意味着你将会编写大量包装方法，它们只是把请求转发给被委托者。这些包装方法既会带来运行时代价，也会带来空间开销，在有些应用中，这可能会有重大影响——甚至会让你无法承受。

与任何技术一样，你必须平衡你的特定应用的各种正面因素和负面因素。在数据库 schema 设计中，常常会为了改善性能而对 schema 进行“反规范化”：违反规范化规则，以换取速度。在这里也可进行类似的折衷。事实上，通过反转得墨忒耳法则，使若干模块紧密耦合，你可以获得重大的性能改进。只要对于那些被耦合在一起的模

物理解耦

在这一节，我们在很大程度上考虑的是怎样进行设计，使系统中的事物保持逻辑上的解耦。但是，随着系统变大，另外有一种相互依赖会变得高度重要：在 *Large-Scale C++ Software Design* [Lak96] 一书中，John Laskos 讨论了与组成系统的文件、目录及库之间的关系有关的各种问题。忽略这些物理设计问题的大型项目最后带来的是以天为单位计算的构建周期、可能会把整个系统作为支持代码拖进来的单元测试、以及其他一些问题。Lakos 先生让人信服地表明，逻辑设计和物理设计必须协同进行——要恢复循环依赖对大量代码造成的损害极其困难。如果你在参与大规模开发，我们向你推荐这本书，即便 C++ 不是你的实现语言。

块而言，这是众所周知的和可以接受的，你的设计就没有问题。

否则，你可能就会发现自己正走在一条通往脆弱、不灵活的未来的道路上，或者，根本没有未来。

相关内容：

- 正交性，34 页
- 可撤消性，44 页
- 按合约设计，109 页
- 怎样配平资源，129 页
- 它只是视图，157 页
- 注重实效的团队，224 页
- 无情的测试，237 页

挑战

- 我们已经讨论了怎样使用委托来使服从得墨忒耳法则更容易，从而减少耦合。但是，编写把调用转发给被委托类所需的所有方法既让人厌烦，又容易出错。编写预处理器、自动生成这些调用的各种优点和缺点是什么？这个预处理器是应该只运行一

次，还是应该用作构建的一部分？

练习

24. 我们在上一页的方框中讨论了物理解耦。下面的 C++ 头文件中，哪一个与系统的其余部分更紧密地耦合在一起？（解答在 293 页）

person1.h:
#include "date.h"

```
class Person1 {
private:
    Date myBirthdate;
public:
    Person1(Date &birthdate);
    // ...
```

person2.h
class Date;

```
class Person2 {
private:
    Date *myBirthdate;
public:
    Person2(Date &birthdate);
    // ...
```

25. 对于下面的以及练习 26 和 27 中的例子，根据得墨忒耳法则，确定所示方法调用是否允许。第一个例子是用 Java 编写的。（解答在 293 页）

```
public void showBalance(BankAccount acct) {
    Money amt = acct.getBalance();
    printToScreen(amt.printFormat());
}
```

26. 这个例子也是用 Java 编写的。（解答在 294 页）

```
public class Colada {
    private Blender myBlender;
    private Vector myStuff;
    public Colada() {
        myBlender = new Blender();
        myStuff = new Vector();
    }
    private void doSomething() {
        myBlender.addIngredients(myStuff.elements());
    }
}
```

27. 这个例子是用 C++ 编写的。（解答在 294 页）

```
void processTransaction(BankAccount acct, int) {
    Person *who;
    Money amt;

    amt.setValue(123.45);
    acct.setBalance(amt);
    who = acct.getOwner();
    markWorkflow(who->name(), SET_BALANCE);
}
```

27 元程序设计

再多的天才也无法胜过对细节的专注。

——Levy's Eighth Law

细节会弄乱我们整洁的代码——特别是如果它们经常变化。每当我们必须去改动代码，以适应商业逻辑，法律或管理人员个人一时的口味的某种变化时，我们都有破坏系统——或引入新 bug——的危险。

所以我们说“把细节赶出去！”把它们赶出代码。当我们在与它作斗争时，我们可以让我们的代码变得高度可配置和“软和”——也就是，容易适应变化。

动态配置

首先，我们想要让我们的系统变得高度可配置。不仅是像屏幕颜色和提示文本这样的事物，而且也包括诸如算法、数据库产品、中间件技术和用户界面风格之类更深层面的选择。这些选择应该作为配置选项，而不是通过集成或工程（engineering）实现。

提示 37

Configure, Don't Integrate

要配置，不要集成

要用元数据（metadata）描述应用的配置选项：调谐参数，用户偏好（user preference），安装目录，等等。

元数据到底是什么？严格地说，元数据是关于数据的数据。最为常见的例子可能是数据库 schema 或数据词典。schema 含有按照名称、存储长度及其他属性，对字段（列）进行描述的数据。你应该能访问和操纵这些信息，就像对数据库中的任何其他数据一样。

我们在其最宽泛的意义上使用该术语。元数据是任何对应用进行描述的数据——应用该怎样运行，它应该使用什么资源，等等。在典型情况下，元数据在运行时，而不是编译时被访问和使用。你每时每刻都在使用元数据——至少你的程序是这样。假

定你点击某个选项，隐藏你的 Web 浏览器上的工具栏，浏览器将把该偏好作为元数据存储在某种内部数据库中。

这个数据库可以使用私有格式，也可以使用标准机制。在 Windows 下，初始化文件（使用后缀.ini）或系统注册表中的条目都很典型。在 Unix 下，X Window System 使用 Application Default 文件提供类似的功能。Java 使用的是 Property 文件。在所有这些环境中，你通过指定关键字来获取值。另外，更强大和灵活的元数据实现会使用嵌入式脚本语言（详情参见“领域语言”，57 页）。

Netscape 浏览器实际上使用了这两种技术实现偏好。在版本 3 中，偏好被存为简单的键/值对：

```
SHOW_TOOLBAR: False
```

后来，版本 4 的偏好看起来更像是 JavaScript：

```
user_pref("custtoolbar.Browser.Navigation_Toolbar.open", false);
```

元数据驱动的应用

但我们不只是想把元数据用于简单的偏好。我们想要尽可能多地通过元数据配置和驱动应用。我们的目标是以声明方式思考（规定要做什么，而不是怎么做），并创建高度灵活和可适应的程序。我们通过采用一条一般准则来做到这一点：为一般情况编写程序，把具体情况放在别处——在编译的代码库之外。

提示 38

Put Abstractions in Code, Details in Metadata

将抽象放进代码，细节放进元数据

这种方法有若干好处：

- 它迫使你解除你的设计的耦合，从而带来更灵活、可适应性更好的程序。
- 它迫使你通过推迟细节处理，创建更健壮、更抽象的设计——完全推迟到程序之外。

- 无需重新编译应用，你就可以对其进行定制。你还可以利用这一层面的定制，轻松地绕开正在运行的产品系统中的重大 bug。
- 与通用的编程语言的情况相比，可以通过一种大为接近问题领域的方式表示元数据（参见“领域语言”，57 页）。
- 你甚至还可以用相同的应用引擎——但是用不同的元数据——实现若干不同的项目。

我们想要推迟大多数细节的定义，直至最后时刻，并且尽可能让细节保持“柔和”——尽可能易于改动。通过精心制作允许我们快速作出变更的解决方案，我们将能够更好地应对使许多项目覆没的“转向”（directional shift）（参见“可撤销性”，44 页）。

商业逻辑

于是你让数据库引擎的选择变成了配置选项，并提供了元数据来确定用户界面风格。我们还能做得更多吗？肯定能。

因为与项目的其他方面相比，商业政策与规则更有可能发生变化，以一种非常灵活的格式维护它们很有意义。

例如，你的采购应用可能包括了各种企业政策。也许你在 45 日内向小供应商付款，在 90 日内向大供应商付款。让供应商类型的定义以及时间周期自身，成为可配置的。抓住机会实行一般化。

也许你在编写一个具有可怕的工作流需求的系统。动作会根据复杂的（和变化的）商业规则启动和停止。考虑在某种基于规则的系统（即专家系统）中对它们进行编码，并嵌入到你的应用中。这样，你将通过编写规则、而不是修改代码来配置它。

对于不那么复杂的逻辑，可以使用小型语言加以表达，从而消除在环境变化时重新编译和重新部署的需要。看一看 58 页的例子。

何时进行配置

如“纯文本的力量”（73 页）一节中所提到的，我们建议以纯文本方式表示配置元数据——它会使生活变得容易得多。

但程序应在何时读取该配置？许多程序只在启动时扫描这样的配置，这让人遗憾。如果你需要改变配置，这会迫使你重新启动应用。更为灵活的方法是编写能在运行时重新加载其配置的程序。这一灵活性也有代价：它的实现更复杂。

所以考虑一下你的应用的使用方式：如果它是长期运行的服务器进程，你可以提供某种途径，在程序运行的过程中重新读取并应用元数据。对于能够快速启动的小型客户 GUI 应用，你可能就不需要那样做。

这一现象不只局限于应用代码。我们安装了某个简单应用，或是改动了一个无关紧要的参数，系统就强迫我们重新启动——这样的系统，我们大家都一直觉得很厌烦。

一个例子：Enterprise Java Beans

Enterprise Java Beans (EJB) 是一个用于简化分布式、基于事务的环境中的编程的框架。在此我们提及它，是因为 EJB 说明了元数据可怎样既用于配置应用，又用于降低代码编写的复杂度。

假定你想创建一个 Java 软件，它将参与跨越不同机器、在不同的数据库供应商之间进行、并具有不同的线程及负载平衡模型的事务。

好消息是，你无需担心所有这些事情。你编写一个 *bean*——一个遵循特定约定的自足对象——并将其放置在 *bean container* 中，后者将替你管理大多数低级细节。不用包括任何事务操作或线程管理，你就可以编写 *bean* 代码，EJB 使用元数据来指定事务应怎样被处理。

线程分配和负载平衡是作为容器使用的底层事务服务的元数据指定的。这一分离

给我们提供了极大的灵活性：在运行时动态配置环境

bean 的容器可以用若干不同方式之一替 bean 管理事务（包括让你控制你自己的提交和回滚的一个选项）所有影响 bean 的行为的参数都是在 bean 的部署描述符（deployment descriptor）中指定的——部署描述符是含有我们所需的元数据的序列化对象

像 EJB 这样的分布式系统正在引领我们走向一个可配置的动态系统的新世界。

协作式配置

我们已经讨论了对动态应用进行配置的用户和开发者。但如果你让应用互相配置——让其自身适应其环境的软件——事情又会怎样呢？对已有软件进行未计划的、临时决定的配置，这是一个强大的概念。

操作系统在启动时已经针对硬件进行了配置，而 Web 浏览器会自动用新组件更新自己。

你的更大的系统很可能已经遇到了这样的问题：处理数据的不同版本、以及库和操作系统的不同版次。或许一种更为动态的途径可以帮助你。

不要编写渡渡鸟代码

没有元数据，你的代码就不可能获得它应有的适应性与灵活性。这是一件坏事吗？嗯，在外面的现实世界里，不能适应的物种就会灭亡。

毛里求斯岛上的渡渡鸟不能适应人类和他们的家畜的出现，很久就灭绝了²⁹，这是人类记载的第一起物种的灭绝。

不要让你的项目（或你的职业生涯）走上渡渡鸟的道路。

²⁹ 说定居者用运动球杆打死了这些温和的（读作愚蠢的）的鸟也没用。

相关内容:

- 正交性, 34 页
- 可撤销性, 44 页
- 领域语言, 57 页
- 纯文本的力量, 73 页

挑战

- 针对你目前的项目, 考虑应用有多少内容可以从程序自身移往元数据 所得到的“引擎”看起来会是什么样? 你能否在不同的应用的语境中复用该引擎?

练习

28. 下面的哪些事物最好表示为程序中的代码, 哪些最好表示为外部的元数据?

(解答在 295 页)

- (1) 通信端口指派
- (2) 编辑器对各种语言的语法突显的支持
- (3) 编辑器对不同的图形设备的支持
- (4) 解析器或扫描器的状态机
- (5) 用于单元测试的样本值和结果

28 时间耦合

时间耦合（temporal coupling）是关于什么的，你可能会问——它是关于时间的。

时间是软件架构的一个常常被忽视的方面。吸引我们的时间只是进度表上的时间，发布之前我们剩余的时间——但这不是我们这里在谈论的时间。相反，我们谈论的是，作为软件自身的一种设计要素的时间的角色。时间有两个方面对我们很重要：并发（事情在同一时间发生）和次序（事情在时间中的相对位置）。

我们在编程时，通常并没有把这两个方面放在心上。当人们最初坐下来开始设计架构、或是编写程序时，事情往往是线性的。那是大多数人的思考方式——总是先做这个，然后再做那个。但这样思考会带来时间耦合：在时间上的耦合。方法 A 必须总是在方法 B 之前调用；同时只能运行一个报告；在接收到按钮点击之前，你必须等待屏幕重画。“嘀”必须在“嗒”之前发生。

这样的方法不那么灵活，也不那么符合实际。

我们需要容许并发³⁰，并考虑解除任何时间或次序上的依赖。这样做，我们可以获得灵活性，并减少许多开发领域中的任何基于时间的依赖：工作流分析、架构、设计、还有部署。

工作流

在许多项目中，我们需要把用户的工作流当作需求分析的一部分来进行建模和分析。我们想要找出在同一时间可能发生什么，以及什么必须以严格的次序发生。要做到这一点，一种途径是使用像 UML 活动图（UML activity diagram）³¹这样的表示法来捕捉他们对工作流的描述。

活动图由一组绘制成圆角方框的动作组成。离开某个动作的箭头或者去往另一个

³⁰ 在此我们不会深入讨论并发或并行编程的细节；好的计算机科学课本应该涵盖相关的基础内容，包括调度、死锁、饥饿、互斥/信号量，等等。

³¹ 更多关于所有 UML 图类型的信息，参见[FS97]。

动作(一旦第一个动作完成,后者就可以开始),或者去往叫做同步条(synchronization bar)的粗线。一旦去往某个同步条的所有动作完成,你就可以继续沿着任何离开同步条的箭头前进。没有箭头进入的活动可在任何时候开始。

你可以使用动作图,通过找出本来可以、但却没有并行执行的动作,使并行度最大化。

提示 39

Analyze Workflow to Improve Concurrency

分析工作流,以改善并发性

例如,在我们的搅拌机项目中(练习 17, 119 页),用户最初可能会这样描述他们目前的工作流:

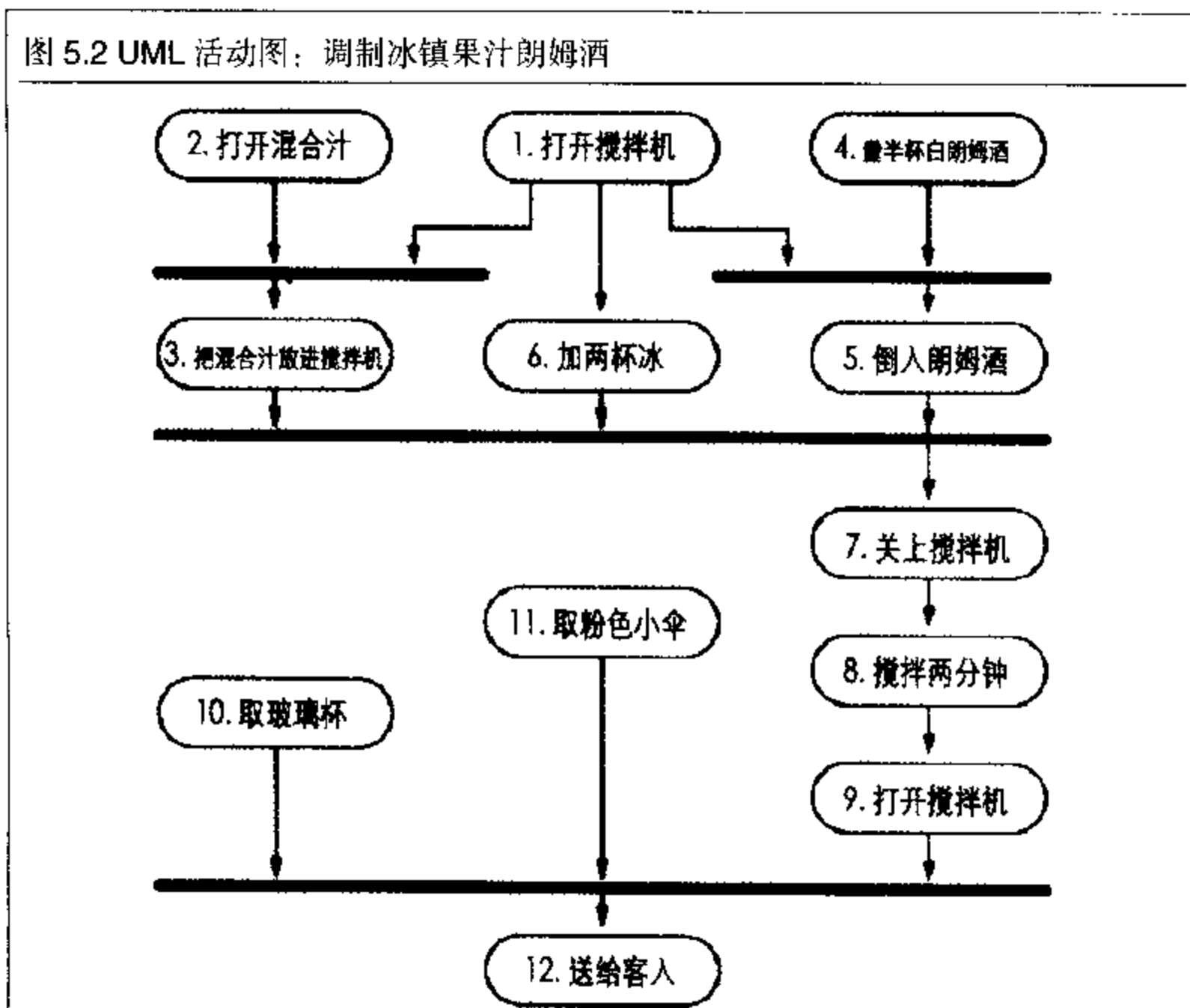
1. 打开搅拌机
2. 打开菠萝和椰子混合汁
3. 把混合汁放进搅拌机
4. 量半杯白朗姆酒
5. 倒入朗姆酒
6. 加两杯冰
7. 关上搅拌机
8. 搅拌两分钟
9. 打开搅拌机
10. 取玻璃杯
11. 取粉色小伞
12. 送给客人

尽管他们逐次描述这些动作,甚至可能会逐次完成它们,我们注意到许多动作都可以并行进行,如下一页图 5.2 中的活动图所示。

看到依赖实际存在于何处,可能会对我们有所启发。在这个例子中,顶层的任务(1、2、4、10 和 11)都可以在前面同时发生。任务 3、5 和 6 可以随后并行发生。

如果你在参加一个冰镇果汁朗姆酒调制比赛,这些优化可以带来极大的效果。

图 5.2 UML 活动图：调制冰镇果汁朗姆酒



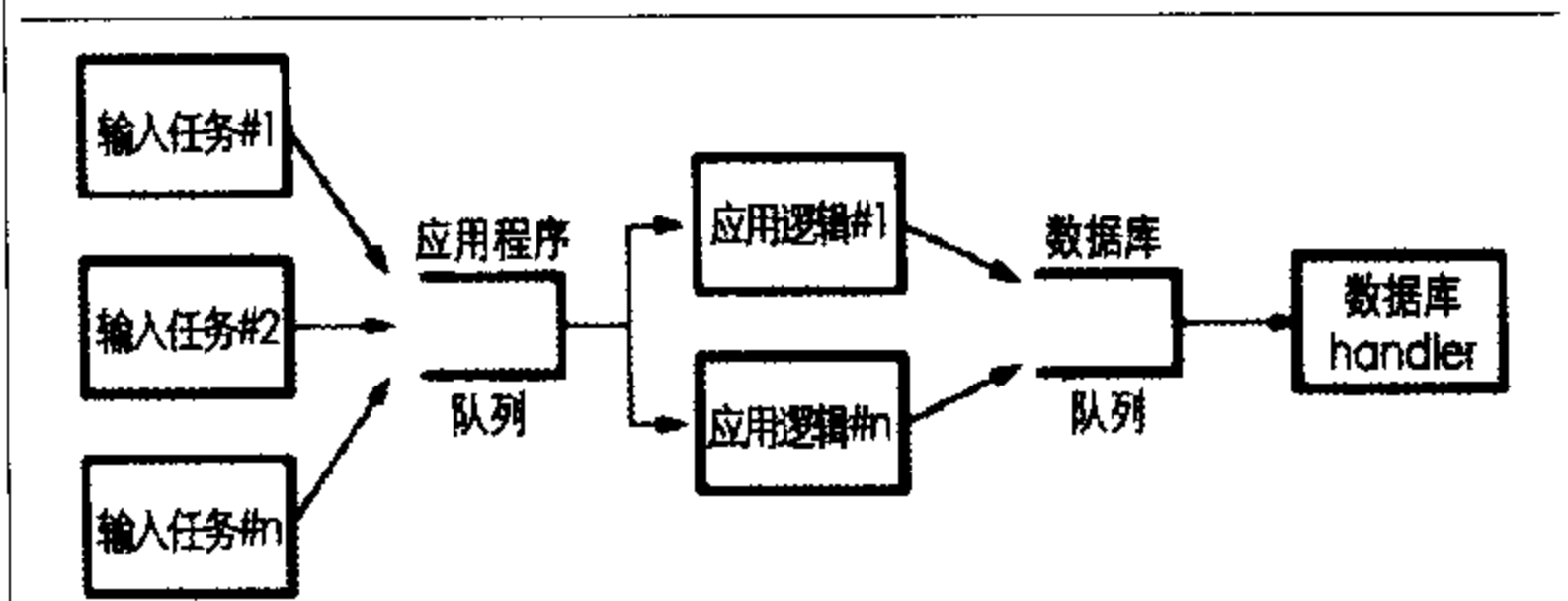
架构

几年前我们编写过一个在线事务处理（OLTP）系统。用最简单的话说，系统所必须做的就是读取请求，依靠数据库对事务进行处理。但我们编写了一个三层的多处理分布式应用：每个组件都是一个独立实体，与所有其他组件一起并发运行。这听起来好像要做更多工作，实情却并非如此：对时间解耦的优势的利用使得它更易于编写。让我们更仔细地看一看这个项目。

该系统从大量数据通信线路上取得请求，并依靠后端数据库对事务进行处理。

该设计处理了以下约束：

图 5.3 OLTP 架构综览



- 数据库操作需要相对较长的时间完成。
- 对于每个事务，在数据库事务被处理的同时，我们不能阻塞通信服务
- 在有太多并发会话时，数据库性能会严重降低
- 多个事务在每条数据线路上并发进行

给了我们最佳性能和最整洁的架构的解决方案如图 5.3 所示。

每个方框表示一个单独的进程；各个进程通过工作队列通信。每个输入进程监控一条通信入线（incoming communication line），并向应用服务器发出请求。所有的请求都是异步的：一旦输入进程发出了它的当前请求，它就会回去监控线路上的其他事务。与此类似，应用服务器向数据库进程发出请求³²，并会在每个事务完成时收到通知。

这个例子还说明了在多个消费者进程间进行快速而粗糙的负载平衡的一种途径：饥饿的消费者（hungry consumer）模型。

³² 即使我们在图中把数据库表示为单个的整体式实体，实情并非如此。数据库软件被划分为若干进程和客户线程，但这由数据库软件在内部处理，并非我们的例子的组成部分。

在饥饿的消费者模型中，你用一些独立的消费者任务和一个集中式工作队列取代中央调度器。各个消费者任务从工作队列中抓取一项，并对其进行处理。当各个任务完成其工作时，就回到队列抓取下一项。这样，如果任何特定的任务陷入停顿，其他任务可以利用这一空闲，并且各个组件都可按自己的步伐前进。每个组件都在时间上解除了与其他组件的耦合。

提示 40

Design Using Services
用服务进行设计

实际上我们创建的不是组件，而是服务——位于定义良好的、一致的接口之后的独立、并发的对象。

为并发进行设计

Java 作为一种平台得到了越来越多人的接受，这使得更多的开发者需要面对多线程化编程。但多线程编程施加了某些设计约束——这是一件好事情。那些约束实际上是如此有益，以致于只要我们在编程，我们就应该遵守它们。它们将帮助解除代码的耦合，并与靠巧合编程（参见 172 页）进行斗争。

编写线性代码，我们很容易做出一些假定，把我们引向不整洁的编程。但并发迫使你更仔细地对事情进行思考——这不再是你一个人的舞会。因为事情现在可能会在“同一时间”发生，你可能会突然看到某些基于时间的依赖关系。

首先，必须对任何全局或静态变量加以保护，使其免于并发访问。现在也许是问问你自己，你最初为何需要全局变量的好时候。此外，不管调用的次序是什么，你都需要确保你给出的是一致的状态信息。例如，何时查询你的对象的状态才是有效的？如果你的对象在某些调用之间处在无效状态，你也许就是在依赖一个巧合：没有人会在那个时间点调用你的对象。

假定你有一个窗口子系统，其中的 widget 是先创建，再显示在显示屏上，分两个步骤进行。在其显示出来之前，你不能设置 widget 中的状态。取决于代码的设置方式，你可能会依靠这样一个事实：在你将其显示在屏幕上之前，其他对象都不会使用已创建的 widget。

但这在并发系统中可能并不为真。在被调用时，对象必须总是处在有效的状态中，而且它们可能会在最尴尬的时候被调用。你必须确保，在任何可能被调用的时刻，对象都处在有效的状态中。这一问题常常出现在构造器与初始化例程分开定义的类中（构造器没有使对象进入已初始化状态）。使用在按合约设计（109 页）中讨论过的类不变项有助于让你避开这一陷阱。

更整洁的接口

对并发和时序依赖进行思考还能够引导你设计更整洁的接口。考虑用于把字符串拆分成 token（基本的文法单元——译注）的 C 库例程 strtok。

strtok 的设计不是线程安全的³³，但那还不是最糟的部分：考察一下时间依赖。你必须用你想要解析的变量对 strtok 进行第一次调用，而所有后继调用都要用 NULL。如果你传入非 NULL 值，它就会重新开始对那个缓冲区进行解析。即使不考虑线程，假定你想要用 strtok 同时解析两个不同的字符串：

```
char buf1[BUFSIZ];
char buf2[BUFSIZ];
char *p, *q;

strcpy(buf1, "this is a test");
strcpy(buf2, "this ain't gonna work");

p = strtok(buf1, " ");
q = strtok(buf2, " ");
while (p && q) {
    printf("%s %s\n", p, q);
    p = strtok(NULL, " ");
    q = strtok(NULL, " ");
}
```

³³ 它使用静态数据维护缓冲区中的当前位置。该静态数据没有受到针对并发访问的保护，所以它不是线程安全的。此外，它会连续改动你传入的第一个参数，这可能会带来讨厌的意外。

所示代码不能工作：在各次调用之间，在 `strtok` 中保留有隐含的状态。你同时只能对一个缓冲区使用 `strtok`。

现在在 Java 中，字符串解析器的设计必须与此不同。它必须是线程安全的，并呈现出一致的状态。

```
StringTokenizer st1 = new StringTokenizer("this is a test");
StringTokenizer st2 = new StringTokenizer("this test will work");

while (st1.hasMoreTokens() && st2.hasMoreTokens()) {
    System.out.println(st1.nextToken());
    System.out.println(st2.nextToken());
}
```

`StringTokenizer` 接口要整洁得多，可维护性也更好。它没有包含“意外”，也不会像 `strtok` 那样，可能在未来引发神秘的 bug。

提示 41

Always Design for Concurrency
总是为并发进行设计

部署

一旦你设计了具有并发要素的架构，对许多并发服务的处理进行思考就会变得更容易：模型变成了普遍的。

现在你可以灵活地处理应用的部署方式：单机、客户-服务器，或是 n 层。通过把你的系统架构成多个独立的服务，你也可以让配置成为动态的。通过对并发做出规划，并且解除各个操作在时间上的耦合，你可以拥有所有这些选择——包括单机选择：你可以选择不进行并发。

走另外的路（设法给非并发应用增加并发）要困难得多。如果我们在设计时就考虑了并发，到时我们就可以更容易地满足可伸缩性或性能需求——即使那样的时候不会到来，我们也仍然拥有更整洁的设计带来的好处。

这不是关于时间的吗？

相关内容：

- 按合约设计，109 页
- 靠巧合编程，172 页

挑战

- 当你为上午的工作做准备时，有多少任务是并行进行的？你能在 UML 活动图中加以表达吗？你能找出某种途径，通过提高并发性更快地做好准备吗？

29 它只是视图

那人依然只听到，
他想要听到的东西，
而不顾其他。
啦—啦—啦—

——Simon and Garfunkel, “The Boxer”

很早以前我们就被教导说，不要把程序写成一个大块，而应该“分而治之”，把程序划分成模块。每个模块都有其自身的责任；事实上，模块（或类）的一个好定义就是，它具有单一的、定义良好的责任。

但一旦你基于责任把程序划分成不同模块，你就有了新的问题。在运行时，对象怎样相互交谈？你怎样管理它们之间的逻辑依赖？也就是说，你怎样对这些不同对象中的状态的变化（或数据值的更新）进行同步？这需要以一种整洁、灵活的方式来完成——我们不想让它们互相知道得太多。我们想要每个模块像“The Boxer”歌中的人一样，只听到它想听到的东西。

我们将从事件（event）的概念开始。一个事件就是一条特殊的消息，说明“刚刚发生了某件有趣的事情”（当然，有趣与否在于观看者的眼睛）。我们可以用事件把某

个对象的状态变化通知给可能感兴趣的其他对象。

这样使用事件使得那些对象之间的耦合得以减至最少——事件发送者不需要对接收者有任何明确的了解。事实上，可以存在多个接收者，每个接收者都专注于自己的“议事日程”（很幸运，发送者无需对此有所了解）。

但是，在使用事件时我们需要小心。例如，在 Java 的早期版本中，是由一个例程接收发给某个特定应用的所有事件。这大概不是通往易于维护或易于演化的道路。

发布/订阅

通过单个例程推送所有事件为什么不好？它破坏了对对象封装——现在一个例程必须对许多对象间的交互有密切的了解。它还增加了耦合——而我们正在设法减少耦合。因为对象自身也必须了解这些事件，你很可能还会违反 *DRY* 原则、正交性原则、甚至是日内瓦公约的一些条款。你可能看到过这样的代码——它通常由巨大的 `case` 语句或多路 `if-then` 加以控制。我们可以做得更好。

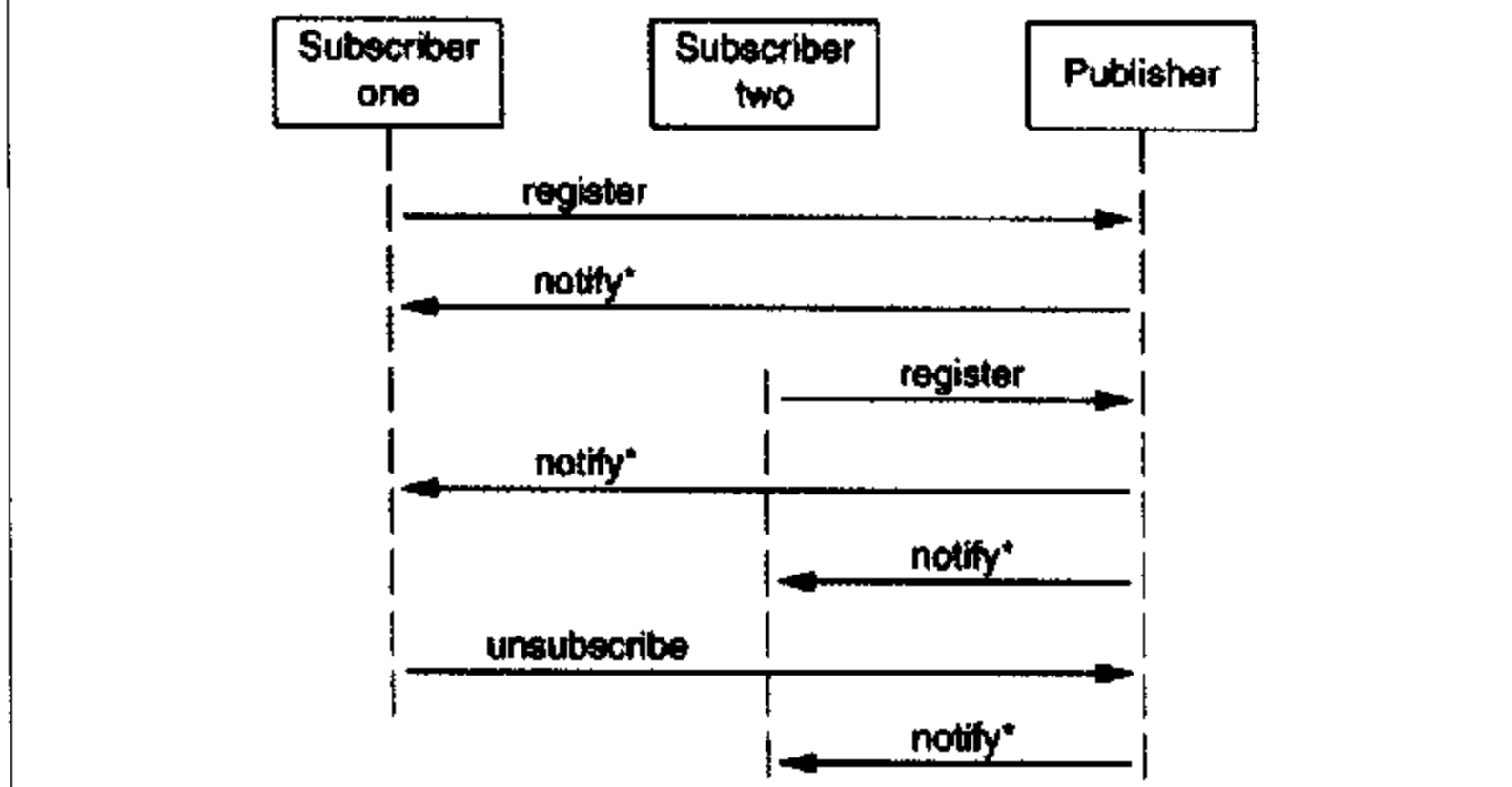
对象应该能进行登记，只接收它们需要的事件，并且决不应该收到它们不需要的事件。我们不想把大量垃圾事件发送给我们对象！相反，我们可以使用一种发布/订阅协议，下一页的图 5.4 使用 UML 序列图阐释了该协议³⁴。

序列图说明若干对象间的消息流，对象按列排列。每个消息用有标记的箭头表示，从发送者所在的列指向接收者所在的列。标记中的星号表明这种类型的消息可以发送不止一次。

如果我们对某个 **Publisher** 生成的特定事件感兴趣，我们所须做的只是登记我们自己。**Publisher** 追踪所有感兴趣的 **Subscriber** 对象；当 **Publisher** 生成有趣的事件时，它将依次调用每个 **Subscriber**，通知它们该事件已经发生。

³⁴ 要获取更多信息，参见 [GHJV95] 中的 Observer 模式。

图 5.4 发布/订阅协议



这个主题有若干变奏——它们反映了其他的通信方式。对象可以按照对等方式（如我们在上面所看到的）使用发布/订阅；也可以使用“软件总线”，由某个中央对象维护侦听器数据库，并适当地分发消息。你甚至可以采用一种把紧急事件广播给所有侦听者的方案——不管它是否做了登记。CORBA Event Service 是分布式环境中一种可能的事件实现的例子，在下一页的方框中将其加以描述。

我们可以使用这种发布/订阅机制实现一种非常重要的设计概念：模型与模型的视图的分离。让我们使用 Smalltalk 的设计，从一个基于 GUI 的例子开始，这种概念就是在 Smalltalk 中诞生的

Model-View-Controller

假定你有一个电子表格应用。除了表格自身中的数值以外，你还有一个图表（graph），把数值显示为柱状图；以及一个在运行的总计对话框，显示表格中的某

列的总和。

CORBA Event Service

CORBA Event Service 允许参与对象通过公共总线（事件信道）发送和接收通知。事件信道仲裁事件处理，并且还解除事件生产者与事件消费者的耦合。它以两种基本方式工作：推和拉。

在推模式中，事件供应者通知事件信道某事件已经发生。信道随即自动把该事件分发给已登记表示感兴趣的所有客户对象。

在拉模式中，客户周期性地轮询事件信道，后者依次又轮询提供与该请求对应的事件数据的供应者。

尽管 CORBA Event Service 可用于实现这一节中讨论的所有事件模型，你还可以将其视为另外一种东西。对于用不同编程语言编写、运行在不同地域的机器上、具有不同架构的对象，CORBA 便利了它们之间的通信。该事件服务位于 CORBA 之上，给了你一种解耦的、与世界各地的应用交互的途径——你从未见过这些应用的编写者，他们用的编程语言你也宁愿不知道。

显然，我们不想拥有三份不同的数据副本。于是我们创建一个模型——数据自身，以及用于对其进行操纵的常用操作。然后我们创建不同的视图，以不同方式显示数据：作为电子表格、作为图表、或是在总计框中。每个视图都有自己的控制器。例如，图表视图可能拥有一个允许你放大、缩小、或是总览数据的控制器。所有这些都不会影响数据自身，而只会影响该视图。

这是位于 Model-View-Controller (MVC) 惯用手法之后的关键概念：既让模型与表示模型的 GUI 分离，也让模型与管理视图的控件分离³⁵。

³⁵ 视图与控制器紧密地耦合在一起，而且在 MVC 的某些实现中，视图与控制器是一个组件。

这样做，你可以利用某些有趣的可能性。你可以支持同一数据模型的多个视图。你可以在许多不同的数据模型上使用公共的查看器。你甚至还可以支持多个控制器，以提供非传统的输入机制。

提示 42**Separate Views from Models**

使视图与模型分离

通过松解模型与视图/控制器之间的耦合，你用低廉的代价为自己换来了许多灵活性。事实上，这种技术是最为重要的维护可撤消性的途径之一（参见“可撤消性”，44页）。

Java 树视图

在 Java 的树 widget 中可以找到 MVC 设计的一个好例子。树 widget（它显示可点击、可遍历的树）实际上是按照 MVC 模式组织的一组不同的类。

要制作功能完备的树 widget，你所要做的只是提供符合 `TreeModel` 接口的数据源。你的代码现在变成了树的模型。

视图由 `TreeCellRenderer` 和 `TreeCellEditor` 类创建，你可以继承并定制它们，在 widget 中提供不同的颜色、字体和图标。`JTree` 充当树 widget 的控制器，并提供了某种常规的查看功能。

因为我们解除了模型与视图的耦合，我们极大地简化了编程。你再也不需要考虑树 widget 编程了。相反，你只需提供数据源就行了。

假定副总裁来到你面前，想要你马上做一个应用，让她浏览存放在大型机的遗留数据库中的公司组织机构图。你只要编写一个读取大型机数据的包装程序，把它作为 `TreeModel` 给出，就成了：你有了一个完全可以浏览的树 widget。

现在你可以变点花样，开始使用查看器类：你可以改变节点的绘制方式，使用特

殊的图标、字体、或是颜色。当副总裁回来，说新的公司标准要求针对特定的雇员使用骷髅图标，你可以对 `TreeCellRenderer` 做出改动，而不用修改其他任何代码。

超越 GUI

尽管在典型情况下，MVC 是在 GUI 开发的语境中教授的，它其实是一种通用的编程技术。视图是对模型（也许是其子集）的一种解释——它无需是图形化的。控制器更是一种协调机制，不一定要与任何种类的输入设备有关。

- **模型**。表示目标对象的抽象数据模型。模型对任何视图或控制器都没有直接的了解。
- **视图**。解释模型的方式。它订阅模型中的变化和来自控制器的逻辑事件。
- **控制器**。控制视图、并向模型提供新数据的途径。它既向模型、也向视图发布事件。

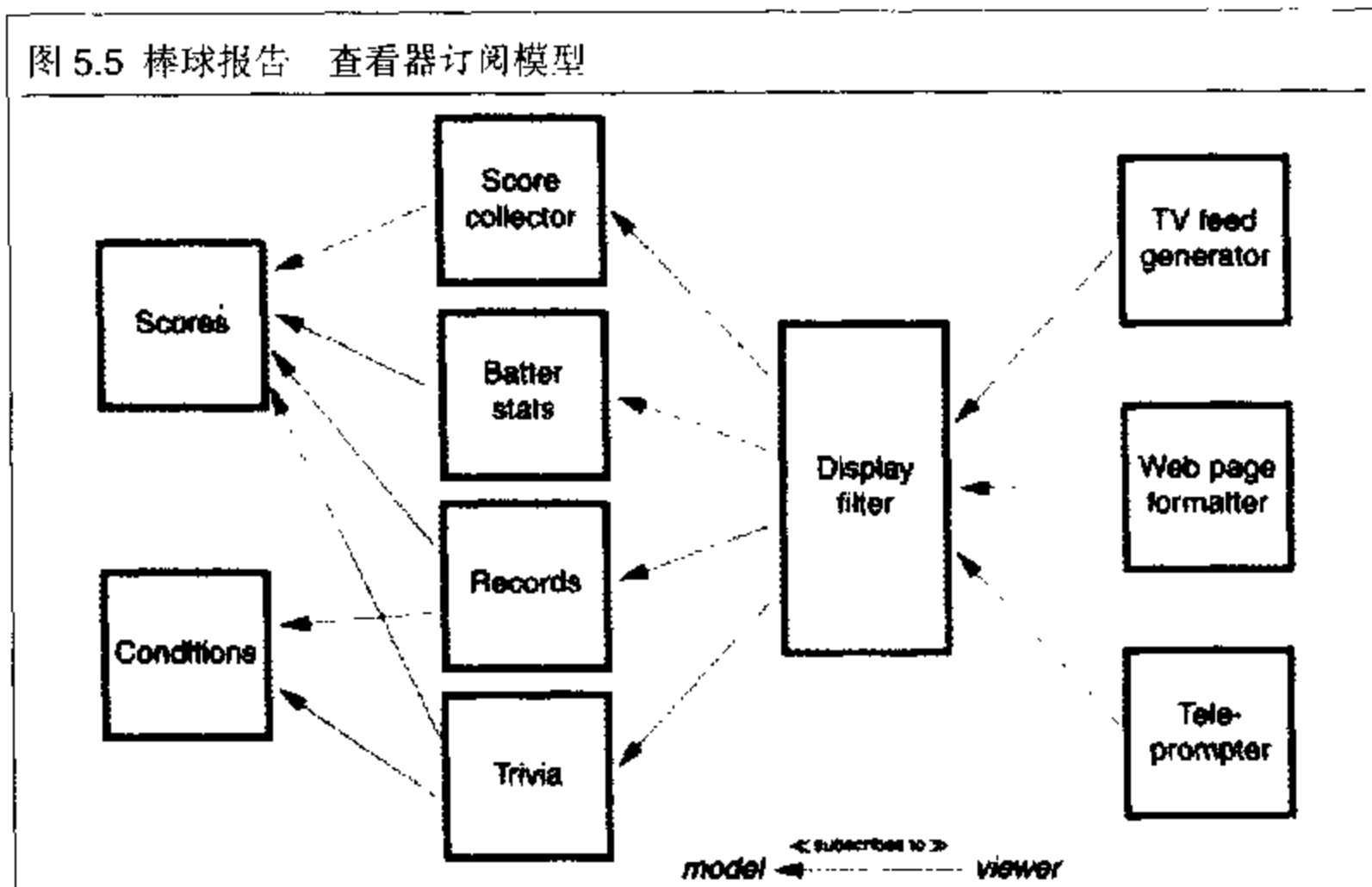
让我们看一个非图形例子。

棒球是一种独特的风俗。你还能在哪里听到这样精彩的闲谈：“这场比赛已经成为在周二、在雨中、在人造灯光下、在队名都是以元音开头的两个球队之间进行的比分最高的比赛”？假定我们接到指令，要开发软件支持那些必须负责地报告比分、统计信息和闲话的坚忍不拔的解说员。

显然我们需要关于正在进行的比赛的信息——参赛球队、各种状况、正在击球的球员、比分，等等。这些事实构成了我们的模型；它们将在新信息到达时更新（更换投手、球员出局、下雨了……）。

随后我们将拥有一些使用这些模型的视图对象。某个视图可以查看垒得分，以更新当前比分。另一个视图可以接收新击球手通知，并获取他们的本年度统计信息的简要汇总。第三个可以查看数据，检查是否有新的世界记录。我们甚至可以有一个琐事查看器，负责报告那些让观众发抖的怪诞而无用的事实。

图 5.5 棒球报告 查看器订阅模型



但我们不想直接用所有这些视图淹没可怜的解说员。相反，我们将让每个视图生成“有趣”事件的通知，并让某个更高级的对象安排要显示的内容³⁶。

这些查看器对象突然变成了更高级对象的模型，后者自己可能又是不同的格式化查看器的模型。某个格式化查看器可以为解说员创建提词机脚本，另一个可以在卫星上行链路上直接生成视频字幕，还有一个可以更新网络或球队的网页（参见图 5.5）。

这种模型——查看器网络是一种常用的（也是很有价值的）设计技术。每条链路都解除原始数据与创建它的事件的耦合——每个新的查看器都是一种抽象。而且因为各种关系是一个网络（而不仅仅是线性链），我们获得了许多灵活性。每个模型都可以有许多视图，一个查看器也可以与多个模型一起工作。

³⁶ 一架飞机从我们头顶飞过也许并不有趣，除非它是当晚从我们头顶飞过的第 100 架飞机。

在这样的高级系统中，拥有调试视图——向你显示模型深层细节的专用视图——可能会很方便。增加一种设施、用于跟踪各个事件，也可以节省大量时间。

（在这么多年之后）仍然有耦合

尽管我们在减少耦合方面取得了进展，侦听者和事件生成者（订阅者和发布者）相互间仍有一些了解。例如，在 Java 中，它们必须就公共接口定义和调用约定达成一致。

在下一节中，我们将考察通过使用某种形式的发布和订阅，进一步减少耦合的途径，在其中各参与方全都无须互相了解，也无须互相直接调用。

相关内容：

- 正交性，34 页
- 可撤销性，44 页
- 解耦与得墨忒耳法则，138 页
- 黑板，165 页
- 全都是写，248 页

练习

29. 假定你有一个航空订座系统，其中包含有这样的航班概念：（解答在 296 页）

```
public interface Flight {  
    // Return false if flight full.  
    public boolean addPassenger(Passenger p);  
    public void addToWaitList(Passenger p);  
    public int getFlightCapacity();  
    public int getNumPassengers();  
}
```

如果你把乘客加入等候名单，他们将在有空座时被自动加入该航班。

有一个大型的报表作业负责查看超额订座或是满员的航班，以提议何时增开航班。它工作得很好，但运行时间却需要数小时。

我们想在处理等候名单的时候多一点灵活性，而且我们必须对那个大报表做点什么——它的运行时间太长了。使用这一节介绍的思想重新设计这个接口。

30 黑板

字迹在墙上……

你通常可能不会把优雅与警探联系起来，他们常被陈腐地描绘成某种油炸面圈和咖啡。但考虑一下，侦探会怎样用黑板来协调和处理谋杀案调查。

假定探长一开始就在会议室放置了一个大黑板。在上面，他写下一个问题：

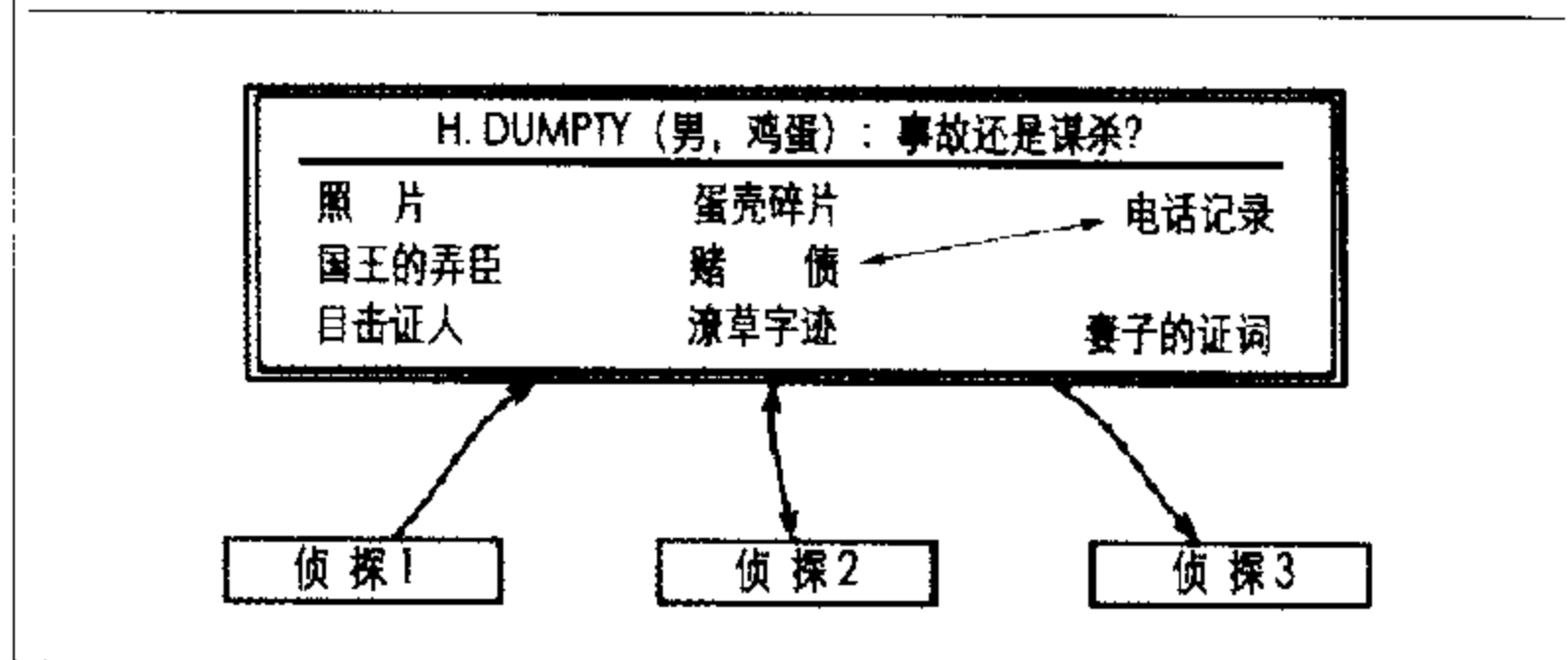
H. DUMPTY (男, 鸡蛋): 事故还是谋杀?

Humpty 真的是掉下去的，还是被推下去的？每位侦探都可以通过增加各种事实、证人陈述以及可能出现的法律证据，等等，为侦破这个神秘的“谋杀”案做出贡献。随着资料的累积，某位侦探可能会注意到某种关联，并张贴他的看法和推断。这个过程持续进行，跨过所有班次，涉及许多不同人员和代理人，直到案件了结。下一页的图 5.6 给出了一个黑板示例。

黑板方法的一些关键特性是：

- 没有侦探需要知道其他任何侦探的存在——他们查看黑板，从中了解新的信息，并且加上他们的发现。
- 侦探可能接受过不同的训练，具有不同程度的教育背景和专业经验，甚至有可能不是在同一管辖区工作。他们都渴望破案，但这就是全部共同点。

图 5.6 有人发现了 Humpty 的赌债和电话记录之间的关联，他也许曾经收到恐吓电话。



- 在这个过程中，不同的侦探可能会来来去去，并且工作班次也可能不同。
- 对放在黑板上的内容没有什么限制——可以是图片、判断、物证，等等

我们参加过的许多项目都涉及 workflow 或分布式数据采集过程。在每个项目中，围绕简单的黑板模型设计解决方案给了我们一个坚实的用于工作的比喻：上面列出的所有关于侦探的特性也同样适用于对象和代码模块。

黑板系统让我们完全解除了我们的对象之间的耦合，并提供了一个“论坛”，知识消费者和生产者可以在那里匿名、异步地交换数据。如你可能会猜想的那样，它还减少了我们必须编写的代码的数量。

黑板实现

基于计算机的黑板系统最初是为在人工智能中应用而发明的，在那样的应用中，要解决的问题大而复杂——语音识别、基于知识的推理系统，等等。

现代的分布式类黑板 (blackboard-like) 系统，比如 JavaSpaces 和 T Spaces[URL

50, URL 52], 以一种键/值对模型为基础; 这种模型首先在 Linda[CG90]中得到推广, 在其中被称为元组空间 (tuple space)

通过这些系统, 你可以在黑板上存放主动的 Java 对象——不只是数据——并且通过对字段进行部分匹配 (通过模板或通配符), 或是通过子类型, 获取这些对象。例如, 假定你有一个类型 **Author**, 是 **Person** 的子类型。你可以使用 **lastName** 值为 “Shakespeare” 的 **Author** 模板搜索含有 **Person** 对象的黑板。你可能会查到叫做 Bill Shakespeare 的作者, 但不会查到叫做 Fred Shakespeare 的园丁。

JavaSpaces 中的主要操作有:

名称	功能
read	在空间中查找并获取数据
write	把数据项放入空间
take	与 read 类似, 但同时从空间中移除该数据项
notify	设定每当写入与模板匹配的对象时就发出通知。

T Spaces 支持一组类似的操作, 但具有不同的名称, 语义也稍有不同。这两个系统都构建得像是数据库产品; 它们为确保数据完整性提供了原子操作和分布式事务。

因为我们可以存储对象, 我们可以使用黑板设计基于对象流, 而不仅仅是数据的算法。就好像我们的侦探可以让证人站到黑板前——证人自己, 而不仅仅是他们的陈述。任何人都可以向证人提出与案情有关的问题、张贴笔录, 并让证人走到黑板的另一个区域, 在那里他也许会做出不同的响应 (如果你允许证人也看黑板)。

像这样的系统的一大优点是, 你与黑板有单一、一致的接口。在构建传统的分布式应用时, 你可能会花大量时间、为系统中的每一个分布式事务和交互精心制作独特的 API 调用。随着接口和交互的组合爆炸, 项目可能很快就会变成噩梦。

组织你的黑板

在侦探们处理大案时，黑板可能会变得混乱，可能会难以在黑板上确定资料的位置。解决方案是对黑板进行分区，并开始以某种方式组织黑板上的资料。

不同的软件系统以不同的方式处理这样的分区，有些使用几乎只有一级的区域或兴趣组，而另一些则采用更加层次化的树状结构。

黑板方式的编程消除了对太多接口的需要，从而能带来更优雅、更一致的系统。

应用实例

假定我们在编写一个程序，接受并处理抵押或贷款申请。辖制这一领域的法律极其复杂，联邦政府、州政府和地方政府都有自己的说法。借贷者必须证明他们已经出示了特定的资料，必须请求提供特定的信息——但又不得提出另外的特定问题，等等、等等。

除了适用法律的迷雾之外，我们还需要应对以下问题：

- 资料到达的次序没有保证。例如，查询信用状况或资格检索可能需要相当长的时间，而像姓名和地址这样的内容可能马上就能获得。
- 资料搜集可能会由位于不同时区、分布在不同办公室的不同的人完成。
- 有些资料搜集可能会由其他系统自动完成。这样的资料也可能会异步到达。
- 然而，特定的资料可能仍会依赖于其他资料。例如，在获得所有权或保险证明之前，你也许不能启动资格检索。

- 新资料的到达可能会引出新的问题和政策。假定信用检查返回的报告不那么有利，现在你需要填写五张额外的表格，也许还要提供血样。

你可以使用工作流系统，设法处理每一种可能的组合和情况。存在许多这样的系统，但它们可能会很复杂，并且需要许多程序员。当规章制度发生变化时，工作流也必须重新组织：人们也许必须改变他们的流程，硬性连接的代码也许必须重写。

黑板，再结合封装了法律需求的规则引擎，是解决这里遇到的困难的一种优雅方案。数据到达的次序无关紧要：在收到某项事实时，它可以触发适当的规则。反馈也很容易处理：任何规则集的输出都可以张贴到黑板上，并触发更为适用的规则。

提示 43

Use Blackboards to Coordinate Workflow

用黑板协调工作流

我们可以用黑板协调完全不同的事实和因素，同时又使各参与方保持独立，甚至隔离。

当然，你可以用更蛮力的方法获得相同的结果，但你得到的将是更脆弱的系统。当它出故障时，国王的所有人马也许都无法使你的程序再工作起来。

相关内容：

- 纯文本的力量，73 页
- 它只是视图，157 页

挑战

- 在现实世界中你是否使用黑板——冰箱旁边的留言板，或是工作用的大白板？是什么使它们有效用？张贴的所有留言的格式是否都一致？这要紧吗？

练习

30. 对于下面的各个应用，黑板系统是否适用？为什么？（解答在 297 页）

- (1) 图像处理** 你想要让一些并行进程抓取图像块，进行处理，并把完成后的图像块放回去
- (2) 机构日程安排** 你的员工分散在全球，在不同的时区，并且说不同的语言，你要安排一次会议
- (3) 网络监控工具** 系统搜集性能统计信息，并收集问题报告。你想要实现一些代理，使用这些信息查找系统中的问题

第 6 章

当你编码时 While You Are Coding

传统智慧认为，项目一旦进入编码阶段，工作主要就是机械地把设计转换为可执行语句。我们认为，这种态度是许多程序丑陋、低效、结构糟糕、不可维护和完全错误的最大一个原因。

编码不是机械工作。如果它是，上世纪 80 年代初期人们寄予厚望的所有 CASE 工具早就取代了程序员。每一分钟都需要做出决策——如果要让所得的程序享有长久、无误和富有生产力的“一生”，就必须对这些决策进行仔细的思考和判断。

不主动思考他们的代码的开发者是在靠巧合编程——代码也许能工作，但却没有特别的理由说明它们为何能工作。在“靠巧合编程”中，我们提倡要更积极地参与编码过程。

尽管我们编写的大部分代码能够快速执行，我们偶尔也会开发出一些算法，可能会让最快的处理器都陷入困境。在“算法速率”中，我们将讨论估算代码的速度的方法，并且还给出一些提示，告诉你怎样在潜在问题发生之前就发现它们。

注重实效的程序员批判地思考所有代码，包括我们自己的。我们不断地在我们的程序和设计中看到改进的余地。在“重构”中，我们将讨论一些即使我们还处在项目中期，也能帮助我们修正现有代码的技术。

只要你在制作代码，你就应当记住，有一天你必须对其进行测试。要让代码易于

测试，这样你将增加它实际通过测试的可能性；我们将在“易于测试的代码”中发展这一思想。

最后，在“邪恶的向导”中，我们建议你小心那些替你编写大量代码的工具，除非你理解它们在做什么。

我们大多数人都能够近乎自动地驾驶汽车——我们不用明确地命令我们的脚踩踏板，或是命令我们的手臂转动方向盘——我们只是想“减速并右转”。但是，可靠的好司机会不断查看周围的情况、检查潜在的问题、并且让自己在万一发生意外时处在有利的位置上。编码也是这样——它也许在很大程度上只是例行公事，但保持警觉能够很好地防止灾难的发生。

31 靠巧合编程

你有没有看过老式的黑白战争片？一个疲惫的士兵警觉地从灌木丛里钻出来。前面有一片空旷地：那里有地雷吗？还是可以安全通过？没有任何迹象表明那是雷区——没有标记、没有带刺的铁丝网、也没有弹坑。士兵用他的刺刀戳了戳前方的地面，又赶紧缩回来，以为会发生爆炸。没有。于是他紧张地向前走了一会儿，刺刺这里，戳戳那里。最后，他确信这个地方是安全的，于是直起身来，骄傲地正步向前走去，结果却被炸成了碎片。

士兵起初的探测没有发现地雷，但这不过是侥幸。他由此得出了错误的结论——结果是灾难性的。

作为开发者，我们也工作在雷区里。每天都有成百的陷阱在等着抓住我们。记住士兵的故事，我们应该警惕，不要得出错误的结论。我们应该避免靠巧合编程——依靠运气和偶然的成功——而要深思熟虑地编程。

怎样靠巧合编程

假定 Fred 接受了一项编程任务。他敲入一些代码，进行试验，代码好像能工作。他又敲入一些代码，进行试验，代码好像还能工作。在进行了几周这样的编码之后，程序突然停止了工作。Fred 花了数小时设法修正它，却仍然不知道原因何在。他可能会花上大量时间四处检查这段代码，却仍然无法修正它。不管他做什么，代码好像就是不能正确工作。

Fred 不知道代码为什么失败，因为他一开始就不知道它为什么能工作。假定进行的是 Fred 所做的有限“测试”，代码好像能工作，但那不过是一种巧合。受到错误信心的鼓动，Fred 冲进了头脑空白的状态。现在，大多数聪明人可能都知道有人像 Fred，但我们更知道——我们不能依靠巧合——对吗？

有时我们可能会依靠巧合。有时要把“幸运的巧合”与有目的的计划混为一谈实在很容易。让我们来看一些例子。

实现的偶然

实现的偶然是那些只是因为代码现在的编写方式才得以发生的事情。你最后会依靠没有记入文档的错误或是边界条件。

假定你用坏数据调用一个例程。例程以一种特定的方式加以响应，而你的代码就以该响应为基础。但原作者并没有预期该例程会以那样的方式工作——它甚至从未被考虑过。当例程被“修正”时，你的代码可能就会出问题。在最为极端的情况下，你调用的例程甚至没有被设计成能做你想要做的事情，但看起来它却工作得很好。以错误的次序、或是在错误的语境中进行调用，是一个与之相关的问题。

```
paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);
```

看来 Fred 在不顾一切地设法把某样东西显示在屏幕上。但这些例程从没有被设计成按这样的方式进行调用；尽管它们看起来能工作，但那实在只是一个巧合。

雪上加霜，当组件终于得以绘制出来，Fred 不会再回去找出似是而非的调用。“它现在能工作了，最好不要再画蛇添足……”

我们很容易被这样的思路愚弄。你为什么要冒把能工作的东西弄糟的风险呢？嗯，我们可以考虑几条理由：

- 它也许不是真的能工作——它也许只是看起来能工作
- 你依靠的边界条件也许只是一个偶然。在不同的情形下（或许是不同的屏幕分辨率），它的表现可能就会不同。
- 没有记入文档的行为可能会随着库的下次发布而变化。
- 多余的和不必要的调用会使你的代码变慢
- 多余的调用还会增加引入它们自己的新 bug 的风险

对于你编写给别人调用的代码，良好的模块化以及把实现隐藏在撰写了良好文档的小接口之后，这样一些基本原则都能对你有帮助。良好制订的合约（参见“按合约设计”，109 页）有助于消除误解。

对于你调用的例程，要只依靠记入了文档的行为。如果出于任何原因你无法做到这一点，那就充分地把你的各种假定记入文档。

语境的偶然

你还可能遇到“语境的偶然”。假定你在编写一个实用模块。只是因为你现在是在为 GUI 环境编写代码，该模块就必须依靠给你的 GUI 吗？你是否依靠说英语的用户？有文化的用户？你还依靠别的什么没有保证的东西？

隐含的假定

巧合可以在所有层面上让人误入歧途——从生成需求直到测试。特别是测试，充满了虚假的因果关系和巧合的输出。很容易假定 X 是 Y 的原因，但正如我们在调试（90 页）中所说的：不要假定，要证明。

在所有层面上，人们都在头脑里带着许多假定工作——但这些假定很少被记入文档，而且在不同的开发者之间常常是冲突的。并非以明确的事实为基础的假定是所有项目的祸害。

提示 44

Don't Program by Coincidence

不要靠巧合编程

怎样深思熟虑地编程

我们想要让编写代码所花的时间更少，想要尽可能在开发周期的早期抓住并修正错误，想要在一开始就少制造错误。如果我们能深思熟虑地编程，那对我们会有所帮助：

- 总是意识到你在做什么。Fred 让事情慢慢失去了控制，直到最后被煮熟，就像“石头汤与煮青蛙”（7 页）里的青蛙一样。
- 不要盲目地编程。试图构建你不完全理解的应用，或是使用你不熟悉的技术，就是希望自己被巧合误导。
- 按照计划行事，不管计划是在你的头脑中，在鸡尾酒餐巾的背面，还是在某个 CASE 工具生成的墙那么大的输出结果上。
- 依靠可靠的事物。不要依靠巧合或假定。如果你无法说出各种特定情形的区别，就假定是最坏的。
- 为你的假定建立文档。“按合约编程”（109 页）有助于澄清你头脑中的假定，并且有助于把它们传达给别人。
- 不要只是测试你的代码，还要测试你的假定。不要猜测；要实际尝试它。编写断言测试你的假定（参见“断言式编程”，122 页）。如果你的断言是对的，你就改善了代码中的文档。如果你发现你的假定是错的，那么就为自己庆幸吧。
- 为你的工作划分优先级。把时间花在重要的方面；很有可能，它们是最难的部分。如果你的基本原则或基础设施不正确，再花哨的铃声和口哨也是没有用的。

- 不要做历史的奴隶 不要让已有的代码支配将来的代码 如果不再适用,所有的代码都可被替换 即使是在一个程序中,也不要让你已经做完的事情约束你下一步要做的事情——准备好进行重构(参见“重构”,184页) 这一决策可能会影响项目的进度 我们的假定是其影响将小于不进行改动造成的影响³⁷

所以下次有什么东西看起来能工作,而你却不知道为什么,要确定它不是巧合

相关内容:

- 石头汤与煮青蛙, 7 页
- 调试, 90 页
- 按合约设计, 109 页
- 断言式编程, 122 页
- 时间耦合, 150 页
- 重构, 184 页
- 全都是写, 248 页

练习

- 31.** 你能否识别出下面的 C 代码段中的一些巧合? 假定这段代码深埋在某个库例程中
(解答在 298 页)

```
fprintf(stderr, "Error, continue?");
gets(buf);
```

- 32.** 这段 C 代码有时在有些机器上能工作,但有时又不能。有什么问题? (解答在 298 页)

```
/* Truncate string to its last maxlen chars */
void string_tail(char *string, int maxlen) {
    int len = strlen(string);
    if (len > maxlen) {
```

³⁷ 这里你也可能会走得太远 我们曾经认识一个开发者,他重写了给他的所有源码,因为他有自己的命名约定

```
strcpy(string, string + (len - maxlen));
}
```

- 33.** 这段代码来自某个通用的 Java 跟踪套件。该函数把一个字符串写到日志文件中。它通过了单元测试, 但当一个 Web 开发者使用它时却失败了。它依靠了什么巧合? (解答在 299 页)

```
public static void debug(String s) throws IOException {
    FileWriter fw = new FileWriter("debug.log", true);
    fw.write(s);
    fw.flush();
    fw.close();
}
```

32 算法速率

在“估算”(64 页)中, 我们对穿过城区所需时间、或是项目完成所需时间这样的事物的估算进行了讨论。但是, 另外有一种估算, **注重实效的程序员**几乎每天都要使用: 估计算法使用的资源——时间、处理器、内存, 等等。

这种估算常常至关重要。给定两种做某事的途径, 你选择哪一种? 你知道在 1 000 条记录的情况下你的程序要运行多久, 但如果增加到 1 000 000 条记录呢? 代码的哪些部分需要优化?

我们发现, 这些问题常常可以通过常识、某种分析、以及叫做“big O”的近似计算表示法来加以回答。

我们说估算算法是什么意思?

大多数并非微不足道的算法都要处理某种可变的输入——排序 n 个字符串、对 $m \times n$ 矩阵求逆、或是用 n 位的密钥解密消息。通常, 这些输入的规模会影响算法: 输入越多, 运行时间就越长, 或者使用的内存就越多。

如果关系总是线性的(于是时间的增加与 n 的值成正比),这一节也就无关紧要了。但是,大多数重要的算法都不是线性的。好消息是大多数算法都是亚线性的。例如,二分查找在查找匹配项时无须查看每一个候选项。坏消息是有一些算法比线性情况要糟得多;其运行时间或内存需求的增长要远远快于 n 。处理 10 个数据项需要 1 分钟的算法要处理 100 个数据项可能需要一生的时间。

我们发现,只要我们编写的是含有循环或递归调用的程序,我们就会下意识地检查运行时间和内存需求。这很少是形式过程,而是快速地确认我们在做的事情在各种情形下是有意义的。但是,有时我们确实会发现自己在进行更为详细的分析。那就是用上 $O()$ 表示法的时候了。

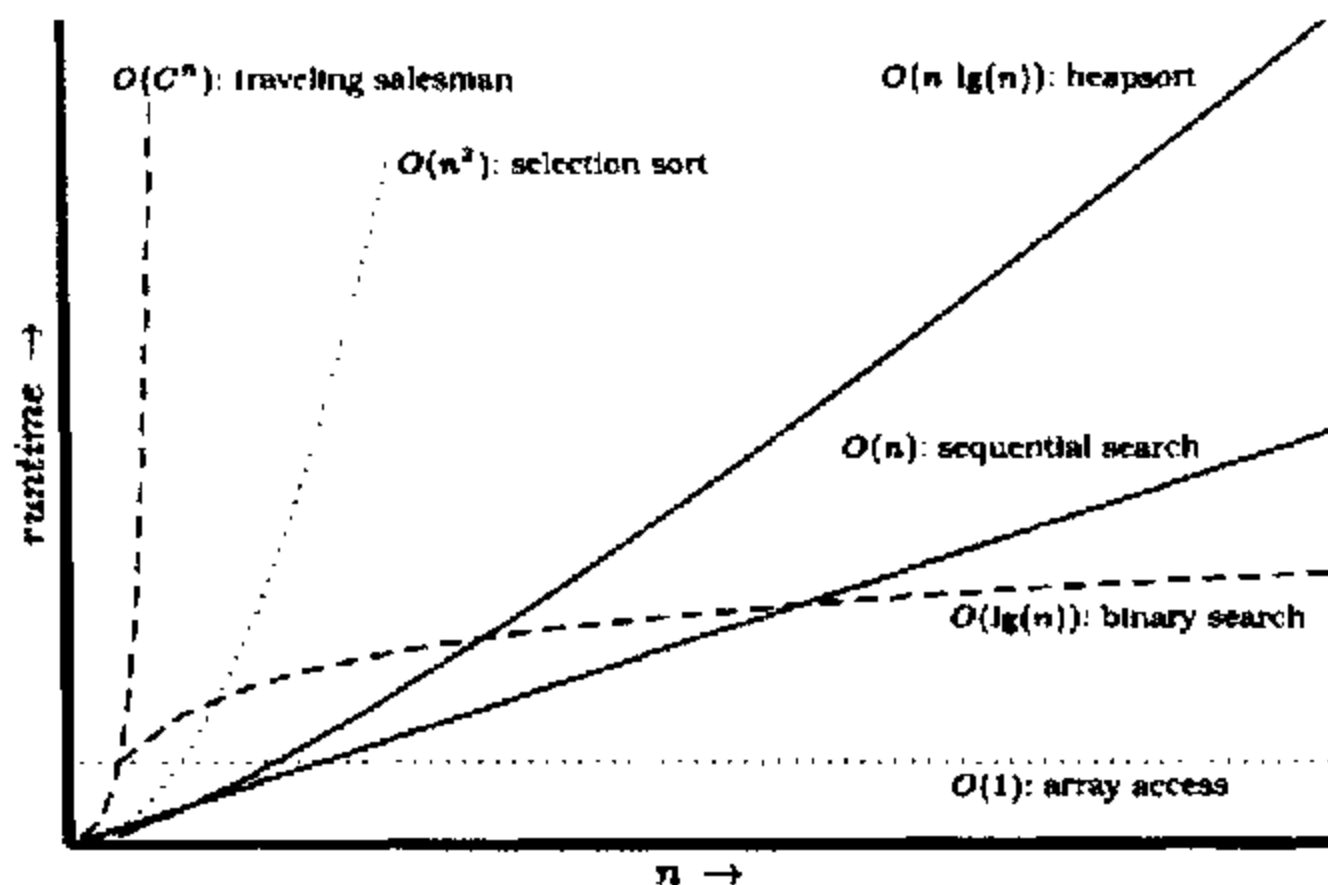
$O()$ 表示法

$O()$ 表示法是处理近似计算的一种数学途径。当我们写下某个特定的排序例程对 n 个记录进行排序所需的时间是 $O(n^2)$ 时,我们的意思是,在最坏的情况下,所需时间随 n 的平方变化。使记录数加倍,时间大约将增加 4 倍。把 O 视为“阶为……”(on the order of)的意思。 $O()$ 表示法对我们在度量的事物(时间、内存,等等)的值设置了上限。如果我们说某函数需要 $O(n^2)$ 时间,那么我们就知道它所需时间的上限不会比 n^2 增长得更快。有时我们会遇到相当复杂的 $O()$ 函数,但因为随着 n 的增加,最高阶的项将主宰函数的值,习惯做法是去掉所有低阶项,并且对任何常数系数都不予考虑。 $O(n^2/2+3n)$ 和 $O(n^2/2)$ 一样,后者又与 $O(n^2)$ 等价。这实际上是 $O()$ 表示法的一个弱点——某个 $O(n^2)$ 算法可能比另一个 $O(n^2)$ 算法要快 1 000 倍,但你从表示法上却看不出来。

图 6.1 给出了你将会遇到的若干常见的 $O()$ 表示法,并且还给出了一个图,比较各个范畴中的算法的运行时间。显然,一旦超过 $O(n^2)$,事情很快就会开始失控。

例如,假定你有一个例程,处理 100 条记录需要 1 秒。处理 1,000 条记录需要多长时间?如果你的代码是 $O(1)$,那么它仍然需要 1 秒。如果是 $O(\lg(n))$,那么你可能要等上 3 秒。 $O(n)$ 将线性增长到 10 秒,而 $O(n\lg(n))$ 大约需要 33 秒。如果你的运气太差,有一个 $O(n^2)$ 例程,那么就坐下来,用上 100 秒等待它完成处理吧。而如果你在

图 6.1 各种算法的运行时间

一些常见的 $O()$ 表示法

$O(1)$	常量型（访问数组元素，简单语句）
$O(\lg(n))$	对数型（二分查找）[$\lg(n)$ 表示法是 $\log_2(n)$ 的简写形式]
$O(n)$	线性型（顺序查找）
$O(n \lg(n))$	比线性差，但不会差很多（快速排序、堆排序的平均运行时间）
$O(n^2)$	平方律型（选择和插入排序）
$O(n^3)$	立方型（ $2n \times n$ 矩阵相乘）
$O(C^n)$	指数型（旅行商问题，集合划分）

使用指数算法 $O(2^n)$ ，你可能会想去煮杯咖啡——你的例程应该要 10^{263} 年才能完成，让我们知道宇宙是怎样终结的。

$O()$ 表示法并非只适用于时间；你可以用它表示算法使用的其他任何资源。例如，能为资源消耗建模，这常常很有用（参见 183 页的练习 35）。

常识估算

你可以使用常识估算许多基本算法的阶。

- **简单循环**。如果某个简单循环从 1 运行到 n ，那么算法很可能就是 $O(n)$ ——时间随 n 线性增加。其例子有穷举查找、找到数组中的最大值、以及生成校验和。
- **嵌套循环**。如果你需要在循环中嵌套另外的循环，那么你的算法就变成了 $O(m \times n)$ ，这里的 m 和 n 是两个循环的界限。这通常发生在简单的排序算法中，比如冒泡排序：外循环依次扫描数组中的每个元素，内循环确定在排序结果的何处放置该元素。这样的排序算法往往是 $O(n^2)$ 。
- **二分法**。如果你的算法在每次循环时把事物集合一分为二，那么它很可能是对数型 $O(\lg(n))$ 算法（参见练习 37，183 页）。对有序列表的二分查找、遍历二叉树、以及查找机器字中的第一个置位了的位，都可能是 $O(\ln(n))$ 算法。
- **分而治之**。划分其输入，并独立地在两个部分上进行处理，然后再把结果组合起来的算法可能是 $O(n \ln(n))$ 。经典例子是快速排序，其工作方式是：把数据划分为两半，并递归地对每一半进行排序。尽管在技术上是 $O(n^2)$ ，但因为其行为在输入的是排过序的输入时会退化，快速排序的平均运行时间是 $O(n \ln(n))$ 。
- **组合**。只要算法考虑事物的排列，其运行时间就有可能失去控制。这是因为排列涉及到阶乘（从数字 1 到 5 有 $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ 种排列）。得出 5 个元素的组合算法所需的时间：6 个元素需要 6 倍的时间，7 个元素则需要 42 倍的时间。其例子包括许多公认的难题的算法——旅行商问题、把东西最优地包装进容器中、

划分一组数，使每一组都有相同的总和，等等。在特定问题领域中，常常用启发式方法（heuristics）减少这些类型的算法的运行时间。

实践中的算法速率

在你的职业生涯中，你不太可能花费大量时间编写排序例程。如果不付出相当的努力，现有库中的例程很可能会胜过你编写的任何东西。但是，我们前面描述的基本的算法类型会不时地再度出现。无论何时你发现自己在编写简单循环，你都知道你有一个 $O(n)$ 算法。如果循环含有内循环，那么就是 $O(m \times n)$ 算法。你应该问问自己，这些值可能有多大。如果数值有限，你就将知道代码要运行多长时间。如果数值取决于外部因素（比如整夜进行的批运行中的记录数，或是名单中的姓名数），那么你可能会想要停下来，考虑一下大量数据对运行时间或内存消耗可能带来的影响。

提示 45

Estimate the Order of Your Algorithms

估算你的算法的阶

你可以通过一些途径处理潜在的问题。如果你有一个 $O(n^2)$ 算法，设法找到能使其降至 $O(n \ln(n))$ 的分而治之的途径。

如果你不能确定代码需要多少时间，或是要使用多少内存，就试着运行它，变化输入记录的数目，或可能影响运行时间的无论什么东西。随后把结果绘制成图。你应该很快就能了解到曲线的形状。随着输入量的增大，它是向上弯曲，是直线，还是保持平直？三个或四个点应该就能告诉你答案。

还要考虑你在代码自身中所做的事情。对于较小的 n 值，简单的 $O(n^2)$ 循环的性能可能会比复杂的 $O(n \ln(n))$ 算法更好，特别是当 $O(n \lg(n))$ 算法有昂贵的内循环时。

在整个理论当中，不要忘了一些实际的考虑。对于小输入集，运行时间看起来也许是在线性增长。但给代码馈入数百万条记录，随着系统开始颠簸，时间就会突然退

化。如果你测试排序例程用的是随机的输入值，当它第一次遇到有序的输入时，你可能会很惊讶。**注重实效的程序员**会设法既考虑理论问题，又考虑实践问题。在进行所有这些估算之后，惟一作数的计时是你的代码运行在实际工作环境中，处理真实数据³⁸时的速率。这引出了我们的下一条提示：

提示 46**Test Your Estimates****测试你的估算**

如果要获得准确的计时很棘手，就用代码剖析器（code profiler）获得你的算法中的不同步骤的执行次数，并针对输入的规模绘制这些数字。

最好的并非总是最好的

你还需要在选择适当算法时注重实效——最快的算法对于你的工作并非总是最好的。假定输入集很小，直截了当的插入排序的性能将和快速排序一样好，而你用于编码和调试的时间将更少。如果你的选择的算法有高昂的设置开销，你也需要注意。对于小输入集，这些设置时间可能会使运行时间相形见绌，并使得算法变得不再适用。

相关内容：

- 估算，64 页

³⁸ 事实上，作者在一台 64MB 的 Pentium 上测试用作本节练习的排序算法，在对 700 多万个数字进行基数排序的过程中，实存被耗尽了。排序例程开始使用交换空间，性能发生了戏剧性的退化。

挑战

- 每个开发者都应该有设计与分析算法的才能。Robert Sedgewick 就这个主题撰写了一系列易于理解的书籍（[Sed83, SF96, Sed92], 等等）。我们建议你在其中选一本加入你的收藏，并记得阅读它。
- 如果想要获得比 Sedgewick 提供的内容更多的细节，可以阅读 Donald Knuth 的权威著作 *Art of Computer Programming*，这本书分析了广泛的算法[Knu97a, Knu97b, Knu98]。
- 在练习 34 中，我们将考察长整数数组的排序。如果键更加复杂，键比较的开销很高，其影响是什么？键结构是否影响排序算法的效率？最快的排序总是最快的？

练习

34. 我们编写了一组简单的排序例程，可从我们的网站（www.pragmaticprogrammer.com）下载。在你可以使用的各种机器上运行它们。你的数字是否遵循预期的曲线？关于你的机器的相对速度，你可以推断出什么？各种编译器优化设置的效果是什么？基数排序真的是线性的吗？（解答在 299 页）
35. 下面的例程打印出二叉树的内容。假定树是平衡的，例程在打印一棵有 1 000 000 个元素的树时大约要使用多少栈空间？（假定子例程调用不会带来显著的栈开销）（解答在 300 页）

```
void printTree(const Node *node) {
    char buffer[1000];
    if (node) {
        printTree(node->left);
        getNodeAsString(node, buffer);
        puts(buffer);
        printTree(node->right);
    }
}
```

36. （除了减小缓冲区）你能否想出任何减少练习 35 中的例程的栈需求的方法？（解答在 300 页）
37. 在 180 页，我们宣称二分法是 $O(\ln(n))$ 。你能证明吗？（解答在 301 页）

38

重构

周遭所见、皆是变易与衰败……

——H. F. Lyte, 《请与我同在》

随着程序的演化, 我们有必要重新思考早先的决策, 并重写部分代码。这一过程非常自然。代码需要演化; 它不是静态的事物。

遗憾的是, 最为常见的软件开发的比喻是修建建筑 (building construction) (Bertrand Meyer[Mey97b]使用了术语“软件建筑”)。但使用建筑作为指导性的比喻暗示了以下步骤:

1. 建筑师 (architect) 绘制出蓝图
2. 承包商挖掘地基、修建上层建筑、布设管线, 并进行最后装修
3. 随后房客高兴地入住, 如有任何问题, 就叫维修人员来修。

可是, 软件的工作方式与此并不怎么相似。与建筑相比, 软件更像是园艺——它比混凝土更有机。你根据最初的计划和各种条件在花园里种植许多花木。有些花木茁壮成长, 另一些注定要成为堆肥。你可能会改变植株的相对位置, 以有效利用光影、风雨的交互作用。过度生长的植株会被分栽或修剪, 颜色不协调的会被移栽到从美学上看更怡人的地方。你拔除野草, 并给需要额外照料的植株施肥。你不断关注花园的兴旺, 并按照需要 (对土壤、植株、布局) 做出调整。

商业人士喜欢修建建筑的比喻: 它比园艺更科学, 它可以重复、具有严格的管理报告层次, 等等。但我们不是在修建摩天大楼——我们不用受物理和现实世界的各种限制的约束。

园艺比喻与软件开发的现实要接近得多。或许特定的例程已变得太大, 或是试图

完成太多事情——它需要被一分为二。没有按照计划完成的事情须要被清除或修剪

重写、重做和重新架构代码合起来，称为重构（refactoring）

你应在何时进行重构

当你遇到绊脚石——代码不再合适，你注意到有两样东西其实应该合并或是其他任何对你来说是“错误”的东西——不要对改动犹豫不决。应该现在就做。无论代码具有下面的哪些特征，你都应该考虑重构代码：

- **重复**。你发现了对 *DRY* 原则的违反（重复的危害，26 页）
- **非正交的设计**。你发现有些代码或设计可以变得更为正交（正交性，34 页）
- **过时的知识**。事情变了，需求转移了，你对问题的了解加深了。代码需要跟上这些变化
- **性能**。为改善性能，你须要把功能从系统的一个区域移到另一个区域。

重构你的代码——四处移动功能，更新先前的决策——事实上是“痛苦管理”（pain management）的一次练习。让我们面对它，四处改动源码可能相当痛苦：它几乎已经在工作，现在事实上却要撕毁了。许多开发者不愿意只是因为代码不完全正确就撕毁代码。

现实世界的复杂情况

于是你去找你的老板或客户，对他们说：“这些代码能工作，但我需要再用一周时间重构它”

我们不能印出他们的回答。

时间压力常常被用作不进行重构的借口。但这个借口并不成立：现在没能进行重构，沿途修正问题将需要投入多得多的时间——那时将需要考虑更多的依赖关系。我们会有更多的时间可用吗？根据我们的经验，没有

你也许可以用一个医学上的比喻来向老板解释这一原则：把需要重构的代码当作是一种“肿瘤”。切除它需要进行“侵入性”的外科手术。你可以现在手术，趁它还小把它取出来。你也可以等它增大并扩散——但那时再切除它就会更昂贵、更危险。等得再久一点，“病人”就有可能会丧命。

提示 47**Refactor Early, Refactor Often****早重构，常重构**

追踪需要重构的事物。如果你不能立刻重构某样东西，就一定要把它列入计划。确保受到影响的代码的使用者知道该代码计划要重构，以及这可能会怎样影响他们。

怎样进行重构

重构肇始于 Smalltalk 社群，并且，与其他趋势（比如设计模式）一道，它已开始赢得更广泛的听众。但作为一个话题，它仍然相当新；没有多少关于它的出版物。第一本关于重构的重要书籍（[FBB+99]，还有[URL 47]），是与本书几乎同时出版的。

就其核心而言，重构就是重新设计。你或你们团队的其他人设计的任何东西都可以根据新的事实、更深的理解、变化的需求，等等，重新进行设计。但如果你无节制地撕毁大量代码，你可能会发现自己处在比一开始更糟的位置上。

显然，重构是一项需要慎重、深思熟虑、小心进行的活动。关于怎样进行利大于弊的重构，Martin Fowler 给出了以下简单提示（参见[FS97]的 30 页的方框）：

1. 不要试图在重构的同时增加功能。
2. 在开始重构之前，确保你拥有良好的测试。尽可能经常运行这些测试。这样，如果你的改动破坏了任何东西，你就能很快知道。

自动重构

在历史上，Smalltalk 用户总是享用着作为 IDE 的组成部分的类浏览器（class browser）。不要把它与 Web 浏览器混为一谈。类浏览器能够让用户浏览并检查类层次和方法。

在典型情况下，类浏览器允许你编辑代码、创建新方法和类，等等。这种思想的下一个变体是重构浏览器（refactoring browser）。

重构浏览器可以半自动地为你进行常见的重构操作：把过长的例程拆成较小的例程、自动传播对方法和变量名的改动、进行拖放以帮助你移动代码，等等。

在我们撰写本书时，这种技术还未在 Smalltalk 以外的世界出现，但这种状况的变化速度很可能会像 Java 的变化速度一样——飞快改变。在此期间，你可以在网上（[URL 20]）找到先驱性的 Smalltalk 重构浏览器。

3. 采取短小、深思熟虑的步骤：把某个字段从一个类移往另一个，把两个类似的方法融合进超类中。重构常常涉及到进行许多局部改动，继而产生更大规模的改动。如果你使你的步骤保持短小，并在每个步骤之后进行测试，你将能够避免长时间的调试。

我们将在“易于测试的代码”（189 页）中进一步讨论这一层面的测试，并在“无情的测试”（237 页）中讨论更大规模的测试，但 Fowler 先生关于维持良好的回归测试的观点是自信地进行重构的关键。

确保对模块做出的剧烈改动——比如以一种不兼容的方式更改了其接口或功能——会破坏构建，这也很有帮助。也就是说，这些代码的老客户应该无法通过编译。于是你可以很快找到这些老客户，并做出必要的改动，让它们及时更新。

所以，下次你看到不怎么合理的代码时，既要修正它，也要修正依赖于它的每样东西。要管理痛苦：如果它现在有损害，但以后的损害会更大，你也许最好一劳永逸地修正它。记住软件的熵（第 4 页）中的教训：不要容忍破窗户。

相关内容

- 我的源码让猫给吃了, 2 页
- 软件的熵, 4 页
- 石头汤与煮青蛙, 7 页
- 重复的危害, 26 页
- 正交性, 34 页
- 靠巧合编程, 172 页
- 易于测试的代码, 189 页
- 无情的测试, 237 页

练习

38. 下面的代码显然在几年里进行了数次更新, 但所做改动并未改善其结构。重构它。

(解答在 302 页)

```
if (state == TEXAS) {
    rate = TX_RATE;
    amt = base * TX_RATE;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
else if ((state == OHIO) || (state == MAINE;
    rate = (state == OHIO) ? OH_RATE : ME_RATE;
    amt = base * rate;
    calc = 2*basis(amt) + extra(amt)*1.05;
    if (state == OHIO)
        points = 2;
}
else {
    rate = 1;
    amt = base;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
```

39. 下面的 Java 类需要支持更多的形状。重构这个类, 为增加做准备。(解答在 303 页)

```
public class Shape {

    public static final int SQUARE    = 1;
    public static final int CIRCLE    = 2;
    public static final int RIGHT_TRIANGLE = 3;

    private int shapeType;
    private double size;
```

```

public Shape(int shapeType, double size) {
    this.shapeType = shapeType;
    this.size = size;
}
// ... other methods ...
public double area(){
    switch (shapeType) {
        case SQUARE:    return size*size;
        case CIRCLE:    return Math.PI*size*size/4.0;
        case RIGHT_TRIANGLE: return size*size/2.0;
    }
    return 0;
}
}

```

40. 这段 Java 代码是将贯穿你的项目使用的某个框架的组成部分 进行重构，让其更一般化，并且将来更易于扩展。（解答在 303 页）

```

public class Window {
    public Window(int width, int height) { ... }
    public void setSize(int width, int height) { ... }
    public boolean overlaps(Window w) { ... }
    public int getArea() { ... }
}

```

34 易于测试的代码

软件 IC 是人们在讨论可复用性和基于组件的开发时喜欢使用的比喻³⁹。意思是软件组件应该就像集成电路芯片一样进行组合。这只有在你使用的组件已知是可靠的时才能行之有效。

芯片在设计时就考虑了测试——不只是在工厂、在安装时，而且也是在部署现场进行测试。更加复杂的芯片和系统可能还拥有完整的 Built-In Self Test (BIST) 特性，用于在内部运行某种基础级的诊断；或是拥有 Test Access Mechanism (TAM)，用以提供一种测试装备，允许外部环境提供激励，并收集来自芯片的响应。

我们可以在软件中做同样的事情。与我们的硬件同事一样，我们也需要从一开始就把可测试性 (testability) 构建进软件中，并且在把各个部分连接在一起之前对每个

³⁹ 术语“软件 IC”（集成电路）似乎是 Cox 和 Novobilski 于 1986 年、在其关于 Objective-C 的书籍 *Object-Oriented Programming*[CN91] 中发明的。

部分进行彻底的测试。

单元测试

硬件的芯片级测试大致等价于软件中的单元测试（unit testing）——在隔离状态下对每个模块进行测试，目的是检验其行为。一旦我们在受控的（甚至是人为的）条件下对模块进行了彻底的测试，我们就能够更好地了解模块在广阔的世界上将怎样起反应。

软件的单元测试是对模块进行演练的代码。在典型情况下，单元测试将建立某种人工环境，然后调用被测试模块中的例程。然后，它根据已知的值，或是同一测试先前返回的结果（回归测试），对返回的结果进行检查。

随后，当我们把我们的“软件 IC”装配进完整系统中时，我们将有信心，各个部分都能够如预期的那样工作，然后我们可以使用同样的单元测试设施把系统当作整体进行测试。我们将在“无情的测试”（237 页）中讨论对系统的大规模检查。

但是，在我们走那么远之前，我们需要决定在单元级测试什么。在典型情况下，程序员会随便把一些数据扔给代码，就说已经测试过了。应用“按合约设计”后面的思想，我们可以做得好得多。

针对合约进行测试

我们喜欢把单元测试视为针对合约的测试（参见“按合约设计”，109 页）。我们想要编写测试用例，确保给定的单元遵守其合约。这将告诉我们两件事情：代码是否符合合约，以及合约的含义是否与我们所认为的一样。我们想要通过广泛的测试用例与边界条件，测试模块是否实现了它允诺的功能。

这在实践中意味着什么？让我们看一看我们在 114 页第一次遇到的平方根例程。其合约很简单：

```
require:
    argument >= 0;
ensure:
    ((result * result) - argument).abs <= epsilon*argument;
```

这告诉了我们要测试什么：

- 传入负数，确定其被拒绝。
- 传入参数 0，确定其被接受（这是边界值）。
- 传入在 0 和可表达的最大参数之间的值，并检验结果的平方与原来的参数之间的差值小于某个值 *epsilon*。

有了这个合约，并假定我们的例程会进行自己的前、后条件检查，我们可以编写一个基本的脚本，演练这个平方根函数

```
public void testValue(double num, double expected) {
    double result = 0.0;

    try {
        // We may throw a
        result = mySqrt(num); // precondition exception
    }
    catch (Throwable e) {
        if (num < 0.0) // If input is < 0, then
            return; // we're expecting the
        else // exception, otherwise
            assert(false); // force a test failure
    }

    assert(Math.abs(expected-result) < epsilon*expected);
}
```

然后，我们可以调用这个例程，测试我们的平方根函数。

```
testValue(-4.0, 0.0);
testValue( 0.0, 0.0);
testValue( 2.0, 1.4142135624);
testValue(64.0, 8.0);
testValue(1.0e7, 3162.2776602);
```

这是一个相当简单的测试；在现实世界中，任何并非微不足道的模块都很可能会依赖其他一些模块，那么，我们该怎样对它们的组合进行测试呢？

假定我们有一个使用 `LinkedList` 和 `Sort` 的模块 `A`。按照顺序，我们会：

1. 全面测试 `LinkedList` 的合约。
2. 全面测试 `Sort` 的合约。
3. 测试 `A` 的合约，它依赖于另外两个合约，但没有直接暴露它们

这种风格的测试要求你首先测试模块的子组件。一旦子模块得到了检验，就可以测试模块自身。

如果 `LinkedList` 和 `Sort` 通过了测试，但 `A` 的测试却失败了，我们可以相当肯定问题是在 `A` 中，或是在 `A` 对这些组件之一的使用中。这一技术是减少调试工作的极好途径：我们可以很快专注于模块 `A` 中可能的问题源，而不用把时间浪费在重新检查其子组件上。

我们为什么要这么费事？最重要的是，我们不想制造“定时炸弹”——呆在周围不被人注意，却在项目后期的尴尬时刻爆炸的东西。通过强调针对合约进行测试，我们可以设法尽可能多地避免那些“下游的灾难”（downstream disaster）。

提示 48

Design to Test
为测试而设计

当你设计模块，甚或是单个例程时，你应该既设计其合约，也设计测试该合约的代码。通过设计能够通过测试，并履行其合约的代码，你可以仔细地考虑边界条件和其他非如此便不会发现的问题。没有什么修正错误的方法比从一开始就避免发生错误更好。事实上，通过在你实现代码之前构建测试，你必须在你确定采用某个接口之前先对它进行试验。

编写单元测试

模块的单元测试不应被扔在源码树的某个遥远的角落里。它们须放置在方便的地方。对于小型项目，你可以把模块的单元测试嵌入在模块自身里。对于更大的项目，我们建议你每个测试都放进一个子目录。不管是哪种方法，要记住，如果你不容易找到它，也就不会使用它。

通过使测试代码易于找到，你是在给使用你代码的开发者提供两样无价的资源：

1. 一些例子，说明怎样使用你的模块的所有功能。

2. 用以构建回归测试、以验证未来对代码的任何改动是否正确的一种手段。

让各个类或模块包含自己的单元测试很方便（但却并非总是可行）例如，在 Java 中，每个类都有自己的 `main`。除了在应用的主类文件里，所有的 `main` 例程都可用于运行单元测试，当应用自身运行时它将会被忽略。这样做的好处是，你交付的代码仍然含有测试，可用于在现场对问题进行诊断。

在 C++ 中，通过使用 `#ifdef` 有选择地编译单元测试，你可以（在编译时）获得同样的效果。例如，这里有一个非常简单的 C++ 单元测试，它嵌入在我们模块中，用与先前定义的 Java 例程类似的 `testValue` 例程检查我们的平方根函数

```
#ifdef _TEST_
int main(int argc, char **argv)
{
    argc--; argv++;          // skip program name

    if (argc < 2) {          // do standard tests if no args
        testValue( 4.0, 0.0);
        testValue( 0.0, 0.0);
        testValue( 2.0, 1.4142135624);
        testValue(64.0, 8.0);
        testValue(1.0e7, 3162.2776602);
    }
    else {                   // else use args
        double num, expected;

        while (argc >= 2) {
            num = atof(argv[0]);
            expected = atof(argv[1]);
            testValue(num, expected);
            argc -= 2;
            argv += 2;
        }
    }
    return 0;
}
#endif
```

这个单元测试或者会运行一个最小测试集，或者，如果给出了参数，允许你从外界传入数据。shell 脚本可以利用这一能力，运行完整得多的测试集。

如果某个单元测试的正确响应是退出，或是中止程序，你该做什么？在这样的情况下，你需要能选择要运行的测试，也许是通过在命令行上指定某个参数。如果你需

要为你的测试指定不同的起始条件，你还需要传入一些参数

但只提供单元测试还不够，你还必须运行它们，并且经常运行它们。如果类偶尔通过了测试，那也是有帮助的。

使用测试装备

因为我们通常都会编写大量测试代码，并进行大量测试，我们要让自己的生活容易一点，为项目开发标准的测试装备（testing harness）。前一节给出的 main 函数是非常简单的测试装备，与之相比，我们通常需要更多的功能。

测试装备可以处理一些常用操作，比如记录状态、分析输出是否符合预期的结果、以及选择和运行测试。装备可以由 GUI 驱动，可以用项目的其他部分所用的语言编写，也可以实现为 makefile 和 Perl 脚本的组合。在练习 41 的解答（305 页）中给出了一个简单的测试装备。

在面向对象的语言和环境里，你可以创建一个提供这些常用操作的基类。各个测试可以对其进行继承，并增加专用的测试代码。你可以使用 Java 中的标准命名约定和反射（reflection），动态地构建测试列表。这一技术是遵循 *DRY* 原则的好方法——你无需维护可用测试的列表。但在你出发去编写自己的装备之前，你可以研究一下 Kent Beck 和 Erich Gamma 的 xUnit（[URL 22]）。他们已经完成了这项艰苦的工作。

不管你决定采用的技术是什么，测试装备都应该具有以下功能：

- 用以指定设置与清理（setup and cleanup）的标准途径。
- 用以选择个别或所有可用测试的方法。
- 分析输出是否是预期（或意外）结果的手段。
- 标准化的故障报告形式。

测试应该是可组合的；也就是说，测试可以由子组件的子测试组合到任意深度。通过这一特性，我们可以使用同样的工具、同样轻松地测试系统的选定部分或整个系统。

即兴测试

在调试过程中，我们可以临时创建一些特定的测试。它们可以像 `print` 语句这样简单，也可以是在调试器或 IDE 环境中交互地输入的一段代码

在调试会话的最后，你需要使即兴测试正式化。如果代码曾经出过问题，它很可能还会再出问题。不要把你创建的测试随便扔掉；把它加到已有的单元测试中

例如，使用 JUnit (xUnit 家族的 Java 成员)，我们可以这样编写我们的平方根测试：

```
public class JUnitExample extends TestCase {

    public JUnitExample(final String name) {
        super(name);
    }

    protected void setUp() {
        // Load up test data...
        testData.addElement(new dblPair(-4.0,0.0));
        testData.addElement(new dblPair(0.0,0.0));
        testData.addElement(new dblPair(64.0,8.0));
        testData.addElement(new dblPair(Double.MAX_VALUE,
                                           1.3407807929942597E154));
    }

    public void testMySqrt() {
        double num, expected, result = 0.0;

        Enumeration enum = testData.element();
        while (enum.hasMoreElements()) {
            dblPair p = (dblPair)enum.nextElement();
            num      = p.getNum();
            expected = p.getExpected();
            testValue(num, expected);
        }
    }

    public static Test suite() {
        TestSuite suite= new TestSuite();
        suite.addTest(new JUnitExample("testMySqrt"));
        return suite;
    }
}
```

JUnit 被设计为可组合的：我们可以随我们的意愿把任意多的测试加到这个套件中，其中每个测试又可以是一个套件。此外，你还可以选择用图形界面或批界面对测

试进行驱动。

构建测试窗口

即使是最好的测试集也不大可能找出所有的 bug；工作环境的潮湿、温暖的状况似乎能把它们从木制品中带出来。

这就意味着，一旦某个软件部署之后，你常常需要对其进行测试——在现实世界的的数据正流过它的血脉时。与电路板或芯片不同，在软件中我们没有测试管脚（test pin），但我们可以提供模块的内部状态的各种视图，而又不使用调试器（在产品应用中这可能不方便，或是不可能）。

含有跟踪消息的日志文件就是这样一种机制。日志消息的格式应该正规、一致，你也许想要自动解析它们，以推断程序所用的处理时间或逻辑路径。格式糟糕或不一致的诊断信息就像是一堆“呕吐物”——它们既难以阅读，也无法解析。

了解运行中的代码的内部状况的另一种机制是“热键”序列。按下特定的键组合，就会弹出一个诊断控制窗口，显示状态消息等信息。你通常不会想把这样的热键透露给最终用户，但这对于客户服务人员却非常方便。

对于更大、更复杂的服务器代码，提供其操作的内部视图的一种漂亮技术是使用内建的 Web 服务器。任何人都可以让 Web 浏览器指向应用的 HTTP 端口（通常使用的是非标准的端口号，比如 8080），并看到内部状态、日志条目、甚至可能是某种调试控制面板。这听起来也许难以实现，其实并非如此。你可以找到用各种现代语言编写、可自由获取、可嵌入的 HTTP Web 服务器。[\[URL 58\]](#)是着手考察 Web 服务器的好地方。

测试文化

你编写的所有软件都将进行测试——如果不是由你和你们团队测试，那就要由最终用户测试——所以你最好计划好对其进行彻底的测试。一点预先的准备可以大大降低维护费用、减少客户服务电话。

尽管有着黑客的名声，Perl 社群对单元测试和回归测试非常认真。Perl 的标准模块安装过程支持回归测试：

```
%make test
```

在这方面 Perl 自身并无任何神奇之处。Perl 使得比较和分析测试结果都变得更为容易，以确保顺应性（compliance）——测试被放在特定的地方，并且有着某种预期的输出。测试是技术，但更是文化；不管所用语言是什么，我们都可以让这样的测试文化慢慢渗入项目中。

提示 49

Test Your Software, or Your Users Will
测试你的软件，否则你的用户就得测试

相关内容

- 我的源码让猫给吃了，2 页
- 正交性，34 页
- 按合约设计，109 页
- 重构，184 页
- 无情的测试，237 页

练习

41. 为练习 17 的解答中描述的搅拌机接口（289 页）设计一个测试用具。编写一个 shell 脚本，对搅拌机进行回归测试。你需要测试基本功能、错误和边界条件以及合约规定的任何义务。对速度改变有何限制？这些限制得到了遵守吗？（解答在 305 页）

35 邪恶的向导

毋庸置疑，应用的编写正变得越来越难。特别是用户界面，正在日益变得复杂。二十年前，一般的应用（如果竟然有界面）会有一个“glass teletype”界面（只使用一个命令行的 CRT 界面——译注）。异步终端在典型情况下会提供字符型交互显示，而可轮询的设备（比如无处不在的 IBM 3270）会让你在按下 **SEND** 键之前输满整个屏幕。现在，用户期望的是图形用户界面，具有语境敏感的帮助、剪贴、拖放、OLE 集成、以及 MDI 或 SDI。用户期望拥有 Web 浏览器集成和瘦客户端支持。

应用自身始终在变得更为复杂。现在的大多数开发都使用多层模型，可能还伴有某种中间件层或事务监控器。这些程序应该是动态的、灵活的，并且能与第三方编写的的应用互操作。

哦，我们有没有提到，我们下周就需要所有这些东西？

开发者正在努力追赶。如果我们使用的工具与 20 年前制作基本的哑终端应用的那些工具一样，我们就永远不会完成任何东西。

于是，工具制造商和基础设施供应商带着魔弹“向导”（wizard）来了。向导很了不起。你需要具有 OLE 容器支持的 MDI 应用？只需点击一个按钮，回答一些简单的问题，向导就会自动为你生成骨架代码（skeleton code）。Microsoft Visual C++ 环境会自动为这样的情况创建超过 1 200 行代码。向导在其他语境中也很勤快。你可以用向导创建服务器组件、实现 Java beans、处理网络接口——所有复杂的、最好有专家帮助的领域。

但使用古鲁（guru）设计的向导不会自动使开发者 Joe 也成为专家。Joe 的感觉可能会相当好——他刚刚制作了大量代码和一个外观相当漂亮的程序。他只要加入具体的应用功能，软件就可以交付了。但 Joe 是在愚弄他自己，除非他真的理解那些替他制作的代码。他是在靠巧合编程。向导是一条“单行道”——它们为你制作代码，然后就走了。如果它们制作的代码不完全正确，或者情形变了，需要你改编代码，你就只能靠自己了。

我们不是在反对向导。相反，我们用了整整一节（代码生成器，102 页）专门讨论怎样编写你自己的向导。但如果你真的使用向导，却不理解它制作出的所有代码，你就无法控制你自己的应用。你没有能力维护它，而且在调试时会遇到很大的困难。

提示 50

Don't Use Wizard Code You Don't Understand

不要使用你不理解的向导代码

有人觉得这是一种极端的看法。他们说，开发者每天都在依赖他们不完全理解的事物——集成电路的量子力学、处理器的中断结构、用于调度进程的算法，系统提供的库里的代码，等等。我们同意。而且如果向导只是开发者可以依赖的一组库调用或标准的操作系统服务，我们对向导的感觉也是一样的。但它们不是，向导生成的代码变成了 Joe 的应用的完整组成部分。向导代码并没有被分解出来，放在整洁的接口后面——它一行一行地与 Joe 编写的功能交织在一起。最后，它不再是向导的代码，而开始变成 Joe 的代码⁴⁰。没有人应该制作他们不完全理解的代码。

相关内容：

- 正交性，34 页
- 代码生成器，102 页

挑战

- 如果你有 GUI 构建向导可用，用它生成一个骨架应用。仔细查看它制作的每行代码。你全部都理解吗？你能否自己制作它？你宁可自己制作它？它是否做了某些你不需要的事情？

⁴⁰ 但是，有一些其他的技术能帮助管理复杂性。我们在正交性（34 页）中讨论了两个：beans 和 AOP。

第 7 章

在项目开始之前 Before The Project

你是否曾经有过你的项目注定要失败的感觉，甚至是在项目启动之前？有时它也许会这样，除非你先建立某些基本准则。否则，也许你现在就可以建议结束它，并且给出资人省下一些钱。

在项目的最开始，你需要确定各种需求。只是听取用户的意见还不够，去阅读“需求之坑”，以了解更多的相关信息。

传统智慧和约束管理（constraint management）是“解开不可能解开的谜题”的话题。不管你是在做需求、分析、编码，还是在测试，都会遇到各种难题。大多数时候，它们其实不像最初看起来那么困难。

当你认为你已经解决了问题时，你可能仍然觉得不能启动项目。这只是在拖延，还是别有含义？“等你准备好”将告诉你，何时倾听你头脑里发出的告诫声是谨慎之举。

启动太快是一个问题，但等得太久可能会更糟。在“规范陷阱”中，我们将通过例子讨论规范的各种优点。

最后，我们将在“圆圈与箭头”中考察各种形式开发过程和方法学的一些缺陷。不管它经过了多么周详的考虑，也不管它包括了哪些“最佳实践”，没有什么方法能够取代思考。

在项目启动之前把这些关键问题解决好，你就能更好地避免“分析瘫痪”（analysis paralysis），并实际开始你的成功项目。

36 需求之坑

完美，不是在没有什么需要增加，而是在没有什么需要去掉时达到的。

——Antoine de St. Exupery, *Wind, Sand, and Stars*, 1939

许多书籍和教程都把需求搜集当作项目的早期阶段。“搜集”一词似乎在暗示，一群快乐的分析师，随着背景播放的温柔的《田园交响曲》，寻觅散布在四周地面上的智慧金块。“搜索”暗示着需求已经在那里——你只需找到它们，把它们放进你的篮子，就可以愉快地上路了。

事情在很大程度上并非如此。需求很少存在于表面上。通常，它们深深地埋藏在层层假定、误解和政治手段的下面。

提示 51

Don't Gather Requirements – Dig for Them

不要搜集需求——挖掘它们

挖掘需求

当你在尘土里四处挖掘时，你怎样才能识别出真实的需求？答案既简单又复杂。

简单的回答是，需求是对需要完成的某件事情的陈述。以下陈述是好需求：

- 只有指定人员才能查看员工档案。
- 汽缸盖的温度不能超过临界值，该值因引擎而异。
- 编辑器将突显关键词，这些关键词根据正在编辑的文件的类型确定。

但是，极少有需求像这样明了，这也正是需求分析很复杂的原因

用户可能会这样陈述上面列出的第一条陈述：“只有员工的上级和人事部门才可以查看员工的档案。”这个陈述真的是需求吗？今天也许是，但它在绝对的陈述中嵌入了商业政策。政策会经常改变，所以我们可能并不想把它们硬性地写入我们的需求。我们的建议是，把这些政策的文档与需求的文档分开，并用超链接把两者连接起来，使需求成为一般陈述，并把政策信息作为例子发给开发者——他们需要在实现中支持的事物类型的例子。最后，政策可以成为应用中的元数据。

这是一种相对微妙的区别，但对开发者却有着深远的影响。如果需求被陈述为“只有人事部门才能查看员工档案”，开发者最后就可能会编写代码，在每次应用访问这些文件时进行明确的检查。但是，如果陈述是“只有得到授权的用户可以访问员工档案”，开发者就可能会设计并实现某种访问控制系统。当政策改变时（它会改变的），只有该系统的元数据需要更新。事实上，以这样的方式搜集需求会很自然地让你去开发为支持元数据而进行了良好分解的系统。

在讨论用户界面时，需求、政策和实现之间的区别可能会变得非常模糊。“系统必须能让你选择贷款期限”是对需求的陈述。“我们需要一个列表框，以选择贷款期限”可能是，也可能不是。如果用户一定要有列表框，那么它就是需求。相反，如果他们是在描述选择能力，但只是用列表框做例子，这个陈述就可能不是需求。205 页的方框将讨论一个因为忽略用户的界面需要而产生严重问题的项目。

找出用户为何要做特定事情的原因、而不只是他们目前做这件事情的方式，这很重要。到最后，你的开发必须解决他们的商业问题，而不只是满足他们陈述的需求。用文档记载需求背后的原因将在每天进行实现决策时给你的团队带来无价的信息。

有一种能深入了解用户需求、却未得到足够利用的技术：成为用户。你在编写客户服务系统？花几天时间与有经验的支持人员一起接听电话。你正在使人工股票控制

系统自动化? 在交易所里工作一周⁴¹。除了让你洞见系统实际上将如何被使用, 你还会吃惊地发现, “我能否在你们工作时在这里呆上一周?” 这个请求能怎样帮助你与用户建立信任和沟通的基础。只是要记住, 不要妨碍他们的工作!

提示 52

Work with a User to Think Like a User

与用户一同工作, 以像用户一样思考

开采需求的过程也是开始与用户群建立和谐的关系。了解他们对你正在构建的系统的期许和希望的时候。更多的讨论, 见“极大的期望”(255页)。

建立需求文档

于是你同用户一起坐下来, 从他们那里探问真实的需求。你遇到一些合适的, 描述应用需要做什么的情境。作为职业人员, 你会把它们写下来, 并发布每个人都可以用作讨论基础的文档——开发者、最终用户、以及项目出资人。

听众的范围相当广泛。

Ivar Jacobson[Jac94]提出了用于捕捉需求的用例 (use case) 概念。它们让你描述系统的特定用法——不是根据用户界面, 而是以一种更为抽象的方式。遗憾的是, Jacobson 的书在细节上有点模糊, 所以关于用例应该是什么, 现在有许多不同的观点。它是形式的还是非形式的? 是简单的“散文”还是 (像表单一样的) 结构化文档? 什么程度的细节才是合适的 (记住我们有广泛的听众)?

看待用例的一种方式强调其目标驱动 (goal-driven) 的本质。Alistair Cockburn

⁴¹ 一周听起来很长? 其实不然, 特别是如果你观察的处理过程中, 管理人员与工作人员占据着不同的世界时, 管理人员将向你展示事情的运作方式的一种图景。而当你“去到楼下”, 你将会发现非常不同的现实——需要花时间了解的现实。

有时接口就是系统

在 *Wired* 杂志(1999 年 1 月号, 176 页)的一篇文章中, 制作人兼音乐家 Brian Eno 描述了一种不可思议的技术——终极混音台。它能对声音做一切能做的事情。然而, 它并没有让音乐家制作更好的音乐, 或是更快、更低廉地制作唱片, 而是妨碍了他们; 它打扰了创作过程。

要知道为什么, 你必须看一看录音师是怎样工作的。他们凭直觉对声音进行平衡。许多年来, 他们在自己的耳朵和手指之间发展了一种内在的反馈回路——滑动控制器、转动旋钮, 等等。但是, 新混音器的界面却没有利用这些能力。相反, 它迫使使用者在键盘上击键、或是点击鼠标。它提供的功能很全面, 但其包装方却让人感到陌生和怪异。有时, 录音师所需的功能隐藏在晦涩难懂的名称后面, 或是要对基本设施进行不直观的组合才能获得。

那样的环境需要利用已有的各种技能。尽管奴隶般地重复已经存在的事物会阻碍进步, 但我们必须要提供通往未来的过渡。

例如, 通过某种触摸屏界面, 也许可以更好地为录音师服务——仍然能触及, 仍然设置成传统混音台的样子, 同时又允许软件越出固定的旋钮与开关的范围。通过熟悉的界面提供舒适的过渡, 是有助于赢得订单的一种途径。

这个例子还阐释了我们的-一种信念: 成功的工具会适应使用它们的双手。在这个案例中, 你为他人构建的工具必须具有可适应性。

有一篇论文描述了这一途径, 以及能够(严格地或不严格地)当作初始样板的模板([Coc97a], 也可从网上获取: [URL 46])。下一页的图 7.1 给出了他的模板的一个简例, 而图 7.2 给出了他的用例样本。

把形式化的模板用作备忘录, 你可以确保自己包括了用例中所需的所有信息: 履行指标(performance characteristic)、其他有关方面、优先级、频度、以及各种可能突然出现的错误和异常(“非功能需求”)。这也是很好的记录用户意见的地方, 比如“哦, 我们必须做 yyy, 除非我们能满足 xxx 条件”。模板还可以充当和用户会谈用的现成议事安排。

图 7.1 Cockburn 的用例模板

A. 特征信息

- 目标及其语境
- 范围
- 级别
- 前条件
- 成功结束条件
- 失败结束条件
- 主要参与方
- 触发

B. 主要成功情境**C. 扩展****D. 变更****E. 相关信息**

- 优先级
- 履行目标
- 频度
- 上层用例
- 下层用例
- 主要参与方联系渠道
- 协助参与方
- 协助参与方联系渠道

F. 进度表**G. 未决问题**

这种组织方式支持层次结构的用例——在较高级的用例中嵌套更详细的用例。例如，记借方 (post debit) 和记贷方 (post credit) 详细说明了记交易 (post transaction)

用例图

可以用 UML 活动图捕捉 workflow，而且有时要为手边的事务建模，概念层类图很有

图 7.2 用例样本

用例 5: 购货

A. 特征信息

- 目标及其语境: 买方直接向我们公司发出要求, 要求发货和开出账单
- 范围: 公司
- 级别: 概要
- 前条件: 我们知道买方和他们的地址, 等等
- 成功结束条件: 买方收到货物, 我方收到货款
- 失败结束条件: 我方没有发货, 买方没有付款
- 主要参与方: 买方, 代表客户的任何代理 (或计算机)
- 触发: 收到购货要求

B. 主要成功情境

1. 买方打电话要求购货。
2. 公司获取买方名称、地址、欲购货物, 等等
3. 公司向买方提供货物、价格、交货日期等信息
4. 买方签署订单
5. 公司备货, 发送给买方。
6. 公司把发货单寄给买方
7. 买方根据发货单付帐

C. 扩展

- 3a. 公司缺一项货物: 重新协商订单
- 4a. 买方直接用信用卡付款: 接受信用卡付款 (用例 44)
- 7a. 买方退货: 处理退货 (用例 105)

D. 变更

1. 买方可以使用电话、传真、Web 订购表单或电子数据交换
2. 买方可以通过现金、汇款、支票或信用卡付款

E. 相关信息

- 优先级: 最高
- 履行目标: 定货 5 分钟, 45 天内付款
- 频度: 200 单/天
- 上层用例: 管理客户关系 (用例 2)
- 下层用例: 备货 (15)、接受信用卡付款 (44)、处理退货 (105)
- 主要参与方联系渠道: 电话、文件或面谈。
- 协助参与方: 信用卡公司、银行、运输服务公司。

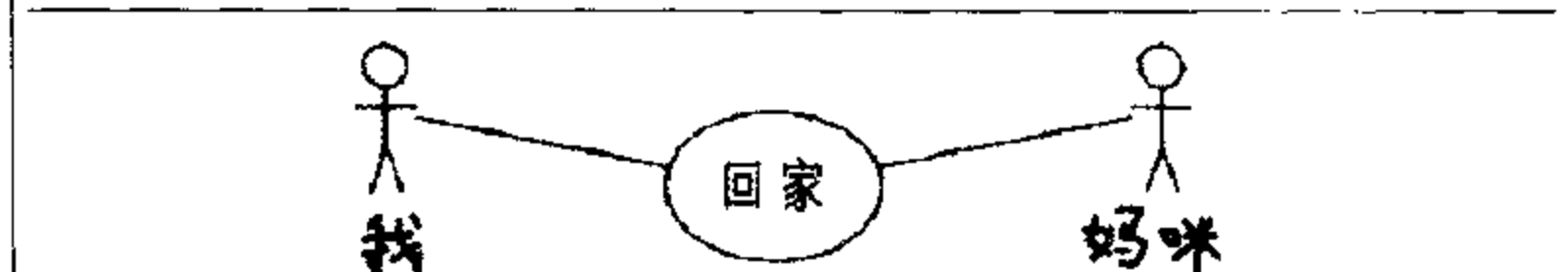
F. 进度表

- 到期日: 1.0 版

G. 未决问题

- 如果我们只有部分货物会怎样?
- 如果信用卡失窃会怎样?

图 7.3 UML 用例——简单得连小孩都会！



用。但真正的用例是具有层次结构和交叉链接的文字描述。用例可以含有指向其他用例的超链接，它们也可以互相嵌套。

有人会认真考虑用像图 7.3 中的那些过分简单化的木棍人来为复杂的信息建立文档，这可能会让我们觉得难以置信。不要做任何表示方法的奴隶；只要是与你的听众交流需求的最好的方法，你都可以加以使用。

规定过度

制作需求文档时的一大危险是太过具体。好的需求文档会保持抽象。在涉及需求的地方，最简单的、能够准确地反映商业需要的陈述是最好的。这并非意味着你可以含糊不清——你必须把底层的语义不变项当作需求进行捕捉，并把具体的或当前的工作实践当作政策记入文档。

需求不是架构。需求不是设计，也不是用户界面。需求是需要。

看远些

人们常常因 2000 年问题指责短视的程序员，他们在大型机的内存比现代的电视遥控器的内存还少的时代拼命节省几个字节。

但这并不是那些程序员的错，实际上也不是一个内存使用问题。如果说，那也是系统分析师和设计师的错。Y2K 问题的出现有两个主要原因：没有超出当时的商业实践往前看，以及对 *DRY* 原则的违反。

在计算机走上舞台很久以前，商业事务就在使用两个数字长的简捷表示法。那是常见的做法。最早的数据处理应用不过是使已有的商业过程自动化，并且只是重复了这一错误。即使架构要求把两个数字的年份用于数据输入、报表、以及存储，本来也应该设计一种 DATE 抽象，“知道”两个数字的年份只是真实日期的一种缩略形式。

提示 53

Abstractions Live Longer than Details

抽象比细节活得更长久

“看远些”是要求你去预测未来吗？不是。它意味着生成这样的陈述：

系统积极地使用 DATE 抽象。系统将一致、普遍地实现 DATE 服务，比如格式化、存储、以及数学运算。

需求将只规定要使用日期。它也许会示意，可以对日期进行某些数学运算。它也许会告诉你，日期将被存储在各种形式的辅助存储器上。这些陈述是关于 DATE 模块或类的真正的需求。

再抹一层薄薄的薄荷

许多项目的失败都被归咎于项目范围的增大——也称为特性膨胀 (feature bloat)、蔓延特性论 (creeping featurism)、或是需求蔓延 (requirement creep)。这是“石头汤与煮青蛙” (第 7 页) 中的煮青蛙综合症的一个表征。我们要做些什么，才能防止需求悄悄地蔓延到我们身上呢？

在各种文献中，你会找到对许多度量方式 (metrics) 的描述，比如报告与修正的 bug、缺陷密度、内聚、耦合、功能点、代码行数，等等。可以通过手工或软件方式追踪这些度量。

遗憾的是，积极地追踪需求的项目似乎并不是很多。这意味着，它们无法报告范围的变化——谁请求增加新特性、谁批准的、批准的请求总数是多少，等等。

管理需求增长的关键是向项目出资人指出每项新特性对项目进度的影响。当项目已经拖后了一年，各种责难开始纷飞时，能够准确、完整地了解需求增长是怎样及何时发生的，会很有帮助。

我们很容易被吸进“只是再增加一个特性”的大漩涡，但通过追踪需求，你可以更清楚地看到，“只是再增加一个特性”，其实已经是本月新增的第15个新特性。

维护词汇表

一旦开始讨论需求，用户和领域专家就会使用对他们有特定含义的术语。例如，他们可能会区分“客户”和“顾客”。于是，再在系统中随意使用这两个词就是不合适的。

要创建并维护项目词汇表（project glossary）——这是定义项目中使用的专用术语和词汇的地方。项目的所有参与者，从最终用户到支持人员，都应该使用这个词汇表，以确保一致性。这就意味着，可以访问词汇表的人员范围应该很广泛——这是采用基于 Web 的文档的一个有效论据（过一会还要进一步讨论这一点）。

提示 54

Use a Project Glossary

使用项目词汇表

如果用户和开发者用不同的名称指称同一事物，或是更糟，用同一名称指称不同事物，这样的项目很难取得成功。

把话说出来

在“全都是写”（248页）中，我们将讨论怎样把项目文档发布到内部的网站上，以方便所有参与者的访问。这种分发方法对于需求文档特别有用。

通过把需求制作成超文本文档，我们可以更好地满足不同听众的需要——我们可以给每个读者他们想要的东西。项目出资人可以在高层的抽象层面上巡视，以确保商业目标得以实现。程序员可以使用超链接“钻入”越来越深入的细节中（甚至参考适

当的定义或工程规范)。

基于 Web 的分发方式还能让你避免典型的两英寸厚的文件夹——名为需求分析，其实永远没有人阅读，墨水刚沾上纸面，就过时了。

如果它在 Web 上，连程序员都可以阅读它。

相关内容：

- 石头汤与煮青蛙，7 页
- 足够好的软件，9 页
- 圆圈与箭头，220 页
- 全都是写，248 页
- 极大的期望，255 页

挑战

- 你能否使用你正在编写的软件？如果你自己无法使用该软件，你是否有可能很好地把握需求？
- 选一个你现在需要解决的与计算机无关的问题。为一种非计算机解决方案生成需求

练习

42. 下面各项中，哪些可能是真正的需求？（如果可能）重新陈述那些并非真正的需求的项。（解答在 307 页）

1. 响应时间必须小于 500ms。
2. 对话框背景为灰色。
3. 应用将被组织成一些前端进程和一个后端服务器。
4. 如果用户在数字字段中输入非数字字符，系统将发出蜂鸣声，并且不接受它们。
5. 应用代码和数据必须不超出 256KB。

37

解开不可能解开的谜题

弗里吉亚的国王戈尔迪斯曾经系过一个没有人能解开的结，据说能解开戈尔迪斯结谜题的人将会统治整个亚洲。亚历山大大帝来了，用剑劈开了这个结。只是对要求做了小小的不同解释，就是这样……他后来的确统治了亚洲大部分。

不时地，你会发现自己与一个遇到了非常困难的“谜题”的项目牵扯在一起：某项工程设计你就是找不到头绪，或是你发现有些代码比你想象的要难以编写得多。也许它看起来是不可能解决的，但它真的有看上去那么困难吗？

考虑现实世界的谜题——那些好像是作为圣诞礼物，或是从旧货市场突然出现的形状奇特的小木块、锻铁、或是塑料。你要做的全部事情就是拿掉铁环，或把 T 形块放进盒子里，等等。

于是你拉动铁环，或是试着把 T 形块放进盒子，并且很快发现明显的解决方法没有用。你无法以那样的方式解开谜题，但即使这很明显，人们也仍然会继续尝试同样的事情——一次又一次——并且认为那样肯定能行。

当然不行。解决方法在另外的地方。解开这个谜题的秘诀是确定真正的（而不是想象的）约束，并在其中找出解决方法。有些约束是绝对的；有些则只是先入之见。绝对的约束必须受到尊重，不管它们看上去有多讨厌或多愚蠢。另一方面，有些外表上的约束也许根本不是真正的约束。例如，有一个酒吧里的老把戏：你拿一瓶全新的、未开启的香槟酒，打赌说你可以从中喝出啤酒来。诀窍是把酒瓶倒过来，在瓶底的凹处倒一点啤酒。许多软件问题都可能具有与之相同的欺骗性。

自由度

流行的俗话“在盒子外面思考”鼓励我们找出可能不适用的约束，并忽略它们。

但这个俗语并不完全准确。如果“盒子”是各种约束和条件的边界，那么诀窍就在于找到盒子——它可能比你以为的要大得多。

解开谜题的关键在于确定加给你各种约束，并确定你确实拥有的自由度，因为在其中你将找到你的解决方案。这也是有些谜题为何如此有效的原因：你可能会太快就排除了潜在的解决方案。

例如，你能只用三条直线就把下面的四个点连起来，并且返回起点吗？不能让笔离开纸面，或是折回已经画过的地方[Hol78]。



你必须挑战任何先入之见，并评估它们是否是真实的，必须遵守的约束。

问题并不在于你是在盒子里面思考，还是在盒子外面思考，而在于找到盒子——确定真正的约束。

提示 55

Don't Think Outside the Box – Find the Box

不要在盒子外面思考——要找到盒子

在面对棘手的问题时，列出所有在你面前的可能途径。不要排除任何东西，不管它听起来有多无用或愚蠢。现在，逐一检查列表中的每一项，并解释为何不能采用某个特定的途径。你确定吗？你能否证明？

想一想特洛伊木马——一个棘手问题的新奇解法。你怎样让军队潜入城池，而又不被发现呢？你可以打赌，“走前门”一开始就作为自杀行为而被排除了。

对你的各种约束进行分类，并划定优先级。木匠开始做活路时，会首先锯出最长的木料，然后再从剩下的木头中锯出较小的木料。按照同样的方式，我们想先确定最为严格的约束，然后再在其中考虑其余的约束。

顺便提一下，上面的“四柱谜题”的一种解法将在 307 页给出

一定有更容易的方法！

有时你会发现，自己在处理的问题似乎比你以为的要难得多。感觉上好像是你走错了路——一定有比这更更容易的方法！或许现在你已落在了进度表后面，甚或失去了让系统工作起来的信心，因为这个特定的问题是“不可能解决的”。

这正是你退回一步，问问自己以下问题的时候：

- 有更容易的方法吗？
- 你是在设法解决真正的问题，还是被外围的技术问题转移了注意力？
- 这件事情为什么是一个问题？
- 是什么使它如此难以解决？
- 它必须以这种方式完成吗？
- 它真的必须完成吗？

很多时候，当你设法回答这些问题时，你会有让自己吃惊的发现。很多时候，对需求的重新诠释能让整个问题全都消失——就像是戈尔迪斯结。

你所需要的只是真正的约束、令人误解的的约束、还有区分它们的智慧。

挑战

- 用心想一想你今天遇到的无论什么难题。你能否解开这个戈尔迪斯结？向你自己提出我们在上面列出的那些关键问题，特别是“它必须以这种方式完成吗？”
- 当你签约开发你现在的项目时，是否交给你一组约束？它们是否仍然都适用？对它们的解释是否仍然有效？

38 等你准备好

有时犹豫的人会得以保全。

——James Thurber, *The Glass in the Field*

了不起的表演者有一个共同的特征：他们知道何时开始，何时等待。跳水运动员站在高台上，等待完美的时刻起跳。指挥站在乐队前面，手臂举起，直到她感觉到某个瞬间适于开始演奏。

你是一个了不起的表演者。你也需要倾听内心的低语声：“等一等”。如果你坐下来，开始敲键盘，在你的头脑里反复出现某种疑虑，要注意它。

提示 56

Listen to Nagging Doubts – Start When You're Ready

倾听反复出现的疑虑——等你准备好再开始

以前有一种网球训练方法，叫做“内在的网球”（inner tennis）。你要花上数小时击球过网，并不特意追求准确性，而是用语言描述球击中的地方与某个目标（常常是一把椅子）的相对位置。其思想是反馈会训练你的下意识和反应能力，于是你的球技就会得到提高，而又无需有意识地了解怎样提高或为何能提高。

作为开发者，你在整个职业生涯中都在做同样的事情。你一直在试验各种东西，看哪些可行，哪些不可行。你一直在积累经验与智慧。当你面对一件任务时，如果你反复感觉到疑虑，或是体验到某种勉强，要注意它。你可能无法准确地指出问题所在，但给它时间，你的疑虑很可能就会结晶成某种更坚实的东西，某种你可以处理的东西。软件开发仍然不是科学。让你的直觉为你的表演做出贡献。

是良好的判断，还是拖延

每个人都害怕空白的纸页。启动新项目（甚或是已有项目中的新模块）可能会是

让人身心交瘁的经验。我们许多人更愿意延缓做出最初的启动承诺。那么，你怎样才能知道，你什么时候是在拖延，而不是在负责地等待所有工作准备就绪？

在这样的情形下，我们采用的一种行之有效的技术是开始构建原型。选择一个你觉得会有困难的地方，开始进行某种“概念验证”（proof of concept）。在典型情况下，可能会发生两种情况。一种情况是，开始后不久，你可能就觉得自己是在浪费时间。这种厌烦可能很好地表明，你最初的勉强只是希望推迟启动。放弃原型，回到真正的开发中。

另一种情况是，随着原型取得进展，你可能会在某个时刻得到启示，突然意识到有些基本的前提错了。不仅如此，你还将清楚地看到可以怎样纠正错误。你将会愉快地放弃原型，投入正常的项目。你的直觉是对的，你为你自己和你的团队节省了可观的、本来会浪费的努力。

当你做出决定，把构建原型当作调查你的不适的一种方法时，一定要记住你为何这样做。你最不想看到的事情就是，你花了几个星期认真地进行开发，然后才想起你一开始只是要写一个原型。

有一点冷嘲热讽的意味：从“政治策略”的角度说，去构建原型也许比简单地宣布“我觉得不该启动”、并开始玩单人纸牌游戏更能让人接受。

挑战

- 与你的同事讨论“启动恐惧症”。他们是否也有同样的经验？他们会加以注意吗？他们用什么诀窍来克服它？团体是能帮助克服个人的勉强，还是会带来压力？

39 规范陷阱

着陆飞行员就是不负责操纵的飞行员，直到“决策高度”（decision altitude）呼叫为止，届时负责操纵的非着陆飞行员把操纵权交给非操纵着陆飞行员，除非后者呼叫“盘旋”，在这种情况下，负责操纵的非着陆飞行员继续操纵，而非操纵着陆飞行员继续不操纵，直到下一次视情况发出“着陆”或“盘旋”呼叫为止。考虑到近来对这些规则的理解多有混乱，我们相信有必要重新对其进行清晰的陈述。

——不列颠航空公司备忘录，摘自 *Pilot Magazine*, 1996.12

编写程序规范就是把需求归约到程序员能够接管的程度的过程。这是一个交流活动，旨在解释并澄清系统的需求，比如消除主要的歧义。除了与最初实现的开发者交谈以外，规范还是留给未来进行代码维护和增强的几批程序员的记录。规范也是与用户的约定——是对他们的需要的汇编，也是一份隐含的合约：最终系统将会符合该合约的要求。

编写规范是一项重要职责。

问题是许多设计者发现很难停下来。他们觉得，他们不应领取每日的薪水，除非每一处小细节都极端详细地确定下来。

这是一个错误，原因如下：首先，认为规范将捕捉系统或其需求的每一处细节和细微差别，这很幼稚。在受限的问题领域中，有一些形式方法能够对系统进行描述，但他们仍然要求设计者向最终用户解释该表示方法的含义——人的解释仍然会搅乱事情。即使这样的解释中的固有问题并不存在，一般用户也很可能无法准确地说出他们所需的系统。他们可能会说，他们已经理解了需求，他们可能会在你制作的 200 页的文档上签字，但你可以确信，一旦他们看到运行的系统，你就会被各种变更要求淹没。

其次，语言自身的表达能力存在着问题。所有的图示技术和形式方法都仍然依赖

于用自然语言表达要进行的操作⁴²。而自然语言实在不能胜任这项工作。看一看任何合约的措词：为了进行精确的表达，律师不得不以最不自然的方式扭曲语言。

这里有一个挑战：写一份简短的描述，告诉别人怎样系鞋带。快，试一试。

如果你和我们还有共同之处，你很可能也会在描述的中途放弃：“现在，转动你的大拇指和食指，让鞋带的自由端在左鞋带底下从里面穿过……”这是一件极其困难的事情。而我们大多数人不用有意识地思考就能系上鞋带。

提示 57

Some Things Are Better Done than Described

对有些事情“做”胜于“描述”

最后，还有“紧身衣效应”。没有给编码者留下任何解释余地的设计剥夺了他们发挥技巧和艺术才能的权利。有人会说，这是出于好意，但他们错了。有些选择常常只有在编码过程中才会显露出来。在编码时，你可能会想：“看那个，因为我用这种方式编写这个例程，我几乎不用什么努力就可以增加这个额外的功能”，或是“规范说要做这个，但我可以通过不同的途径获得几乎完全相同的结果，却只需要一半时间”。显然，你不应该马上就动手修改，但如果你受到规定过细的设计的约束，你甚至不可能发现这些机会。

作为**注重实效的程序员**，你应该倾向于把需求搜集、设计、以及实现视为同一个过程——交付高质量的系统——的不同方面。不要信任这样的环境：搜集需求、编写规范，然后开始编码，所有这些步骤都是孤立进行的。相反，要设法采用无缝的方法：规范和实现不过是同一个过程——设法捕捉和编纂需求——的不同方面。每一步都应该直接流入下一步，没有人为制造的界限。你将会发现，健康的开发过程鼓励把来自实现与测试的意见反馈到规范中。

⁴² 有些形式技术试图用代数方法表达操作，但这些技术很少被用于实践。它们仍然要求分析师向最终用户解释其含义。

澄清一下，我们不是在反对生成规范。相反，我们也发现，有时需要极其详细的规范——出于合约的原因、因为你工作的环境、或是因为你正在开发的产品性质⁴³。只是要知道，随着规范越来越详细，你得到的回报会递减，甚至会负回报。如果没有任何支持实现（*supporting implementation*），或是没有构建原型，在规范之上构建另外一层规范时也要小心；规定无法构建的东西太容易了。

你越是把规范当作安乐毯，不让开发者进入可怕的编码世界，进入编码阶段就越困难。不要掉进这样的规范螺旋中：在某个时刻，你需要开始编码！如果你发现你的团队全都包裹在暖和、舒适的规范中，把他们赶出来。考虑构建原型，或是考虑曳光弹开发。

相关内容：

- 曳光弹，48 页

挑战

- 上文中提到的鞋带例子是对书面描述的问题的有趣阐释。你考虑过用图而不是文字描述该过程吗？照片？某种来自拓扑学的形式表示法？金属丝模型？你会怎样教初学走路的幼儿？

有时一幅图胜过千言万语，有时却又毫无用处。如果你发现自己规定过度，图片或特殊的表示法会有帮助吗？它们必须有多详细？绘图工具何时比白板更好？

⁴³ 详细的规范显然适用于性命攸关的系统。我们觉得给他人使用的接口和库也应该制订详细的规范。如果你的整个产品被视为一组例程调用，你最好确定这些调用得到了良好的规定。

40 圆圈与箭头

(照片上)有圆圈和箭头,每张照片背面有一段话,解释它们各是什么。
这些照片将被用作反对我们的证据……

——Arlo Guthrie, "Alice's Restaurant"

从结构化程序设计开始,经过主程序员团队、CASE 工具、瀑布开发、螺旋模型、Jackson、ER 图、Booch 云、OMT、Objectory、以及 Coad/Yourdon,直到今天的 UML,计算技术从来都不缺少意图使程序设计更像工程的方法。每种方法都聚集了自己的追随者,并且都享受到一段时间的流行。然后一种方法又被下一种方法取代。在所有这些方法当中,或许只有第一种——结构化程序设计——拥有长久的生命。

还有一些开发者,在有許多已沉没项目的大海里漂流,不断抓住最新的时尚,就像是遇到海难的人紧紧抓住漂来的木头一样。每当有新的木头漂过时,他们都会费力地游过去,希望这一块会更好。但到最后,不管漂浮物有多好,这些开发者仍然漫无目的地漂流着。

不要误解我们。我们喜欢(有些)形式技术和方法。但我们相信,盲目地采用任何技术,而不把它放进你的开发实践和能力的语境中,这样的处方肯定会让你失望。

提示 58

Don't Be a Slave to Formal Methods

不要做形式方法的奴隶

形式方法有一些严重缺点:

- 大多数形式方法结合图和某些说明文字来捕捉需求。这些图片代表的是设计者对需求的理解。但是,在许多情况下,这些图对最终用户没有意义,所以设计者不得不对它们进行解释。因此,系统的实际用户并没有对需求进行真正的形式检查——每样东西都以设计者的解释为基础,就像是老式的书面需求一样。我们知道以这种

方式捕捉需求的某些好处，但我们更愿意（在可能的情况下）向用户展示原型，并让他们实际使用

- 形式方法似乎鼓励专门化。一组人构建数据模型，另一组人考察架构，而需求搜集人员则收集用例（或其等价物）。我们看到过这样的方法带来了糟糕的交流和工作的浪费。在设计者和编码者之间，往往还会产生一种“我们 vs. 他们”的心理现象。我们更愿意能了解我们正在开发的整个系统，要深入地了解系统的每一个方面也许不可能，但你应该知道各组件怎样交互、数据存放在哪里、还有需求是什么。
- 我们喜欢编写有适应能力的动态系统，使用元数据让我们在运行时改变应用的特征。现在的大多数形式方法都把静态的对象或数据模型与某种事件或活动图表机制结合在一起。我们还没有见过一种方法，能够让我们阐释我们觉得系统应该展现的动态性。事实上，大多数形式方法会让你误入歧途，鼓励你在对象之间建立静态关系，而这些对象本来应该动态地编织在一起。

形式方法能成功吗

在 1999 年 CACM 的一篇文章[Gla99b]里，Robert Glass 回顾了使用七种不同的软件开发技术（4GL、结构化技术、CASE 工具、形式方法、净室方法学、过程模型、面向对象）改善生产率和质量的研究。他报告说，最初围绕所有这些方法的大肆宣传都是夸张。尽管有迹象表明，有些方法能带来好处，但这些好处只有在下述状况出现之后才会开始显现：在采用该技术、其用户培训自己的同时，生产率和质量显著下降。决不要低估采用新工具和新方法的代价。要做好准备，把使用这些技术的第一个项目当做一种学习经验。

我们应否使用形式方法

绝对应该。但始终要记住，形式开发方法只是工具箱里的又一种工具。如果在仔细分析之后，你觉得需要使用形式方法，那就采用它——但要记住谁是主人。不要变成方法学的奴隶：

圆圈与箭头会让你变成糟糕的主人。**注重实效的程序员**批判地看待方法学，并从各种方法学中提取精华，融合成每个月都在变得更好的一套工作习惯。这至关重要。你应该不断努力提炼和改善你的开发过程。决不要把方法学的呆板限制当做你的世界的边界。

不要向方法的虚假权威屈服。有人也许会带着大量类图和 150 个用例步入会议室，但所有那些纸张仍然只是他们对需求和设计的难免出错的解释。在考察工具的产出时，试着不要考虑它值多少钱。

提示 59

Expensive Tools Do Not Produce Better Designs

昂贵的工具不一定能制作出更好的设计

形式方法在开发中肯定有其位置。但是，如果你遇到一个项目，其哲学是“类图就是应用，其余的只是机械的编码”，你知道，你看到的是一个浸满水的项目团队和一个路途遥远的家。

相关内容：

- 需求之坑，202 页

挑战

- 用例图是 UML 的需求搜集过程的一部分（参见“需求之坑”，202 页）。它们是否是与你用户交流的有效方式？如果不是，你为何在使用它们？
- 你怎么知道某种形式方法在给你的团队带来好处？你能够度量什么？什么构成了改善？你能否区分工具带来的好处和团队成员经验增长带来的好处？
- 将新方法引入你的团队的盈亏平衡点在哪里？在引入新工具时，你怎样评估“将来的好处”与“当前生产率的损失”之间的折衷？
- 适用于大型项目的工具也适用于小型项目？其他方式又如何？

第 8 章

注重实效的项目 Pragmatic Projects

随着你的项目开动，我们需要从个体的哲学和编码问题转向讨论更大的、项目级的问题。我们将不深入项目管理的具体细节，而是要讨论能使项目成功或失败的几个关键区域。

一旦参与项目的人员超过一个，你就需要建立一些基本原则，并相应地分派任务。在“注重实效的团队”中，我们将说明怎样在遵循注重实效哲学的同时做到这一点。

使项目级活动保持一致和可靠的一个最重要的因素是使你的各种工作流程自动化。我们将解释其原因，并在“无处不在的自动化”中给出一些真实的例子。

此前，我们讨论过在你编码的同时进行测试。在“无情的测试”中，我们将进一步讨论项目范围的测试哲学和工具——特别是在你没有大量 QA 人员可以随叫随到时。

与测试相比，开发者更不喜欢的惟一一件事情是撰写文档。不管你是有技术文档撰写者帮助，还是要自己撰写，我们将在“全都是写”中向你说明怎样使这烦人的工作的麻烦少一些，产出多一些。

成功取决于旁观者——项目出资人——的眼睛。重要的是成功的感觉，在“极大的期望”中，我们将向你说明一些能使每个项目的出资人高兴的诀窍。

本书的最后一个提示是其他所有提示的直接结果。在“傲慢与偏见”中，我们鼓励你在作品上签名，并为你所做的事情而自豪。

41 注重实效的团队

在 L 集团，Stoffel 负责监督六名第一流的程序员，其管理上的挑战大概可与放牧猫群相比。

——*The Washington Post Magazine*, 1985 年 6 月 9 日

到目前为止，我们在书中考察的注重实效技术都是要帮助个体成为更好的程序员。这些方法对团队也有效吗？

回答是响亮的“是！”成为注重实效的个体有好处，但如果个体是在注重实效的团队中工作，这些好处就会成倍增长。

在这一节，我们将简要考察怎样把各种注重实效的技术应用于作为整体的团队。这些说明只是一个起点。一旦你有了一组注重实效的开发者，让他们工作在能够发挥自身能力的环境中，他们很快就会发展并提炼他们自己的、有效的团队动力机制。

让我们针对团队，重述前面的部分章节。

不要留破窗户

质量是一个团队问题。最勤勉的开发者如果被派到不在乎质量的团队里，会发现自己很难保持修正琐碎问题所需的热情。如果团队主动鼓励开发者不要把时间花费在这样的修正上，问题就会进一步恶化。

团队作为一个整体，不应该容忍破窗户——那些小小的、无人修正的不完美。团队必须为产品的质量负责，支持那些了解我们在“软件的熵”（4 页）中描述的“不要

留破窗户”哲学的开发者，并鼓励那些还不了解这种哲学的人

在有些团队方法学中有质量官员——团队会把保证产品质量的责任委派给他。这显然是荒谬的：质量只可能源于全体团队成员都做出自己的贡献

煮青蛙

还记得在“石头汤与煮青蛙”（7页）一节中，汤锅里那只可怜的青蛙吗？它没有注意到周围环境的渐变，最终被煮熟了。同样的事情也会发生在不警醒的人身上。在项目开发高涨的热度里，很难再用一只眼睛注意周围的环境。

作为整体的团队甚至更容易被煮熟。大家认为，另外有人在处理某个问题，或是团队领导一定已经批准了用户要求做出的某项改动。即使是目的最明确的团队对项目中的重大改动可能也会很健忘。

与之战斗。确保每个人都主动地监视环境的变化。可以指定一个“首席水情监测员”（chief water tester），让这个人持续地检查范围的扩大、时间标度的缩减、新增特性、新环境——任何不在最初的约定中的东西。对新需求进行持续的度量（参见 209 页）。团队无需拒绝无法控制的变化——你只需注意到它们正在发生。否则，你就会置身于热水中。

交流

显然，团队中的开发者必须相互交谈。我们在 18 页的“交流！”中给出了一些促进交流的建议。但是，人们很容易忘记，团队本身也存在于更大的组织中。团队作为实体需要同外界明晰地交流。

对外界而言，看上去沉闷寡言的项目团队是最糟糕的团队。他们举行无章次的会议，在会上没有人想说话。他们的文档混乱：没有两份文档有相同的外观，每一份都使用不同的术语。

杰出的项目团队有着截然不同的个性。人们希望与他们一同开会，因为他们知道自己将看到准备良好、会让每个人都感到愉悦的演出。他们制作的文档新鲜、准确、一致。团队用一个声音说话⁴⁴。他们甚至还可能有幽默感。

有一个简单的营销诀窍，能帮助团队作为整体与外界交流：创立品牌。当你启动项目时，给它取一个名字，最好是不寻常的某种东西。（过去，我们给项目取过像“捕食绵羊的杀手鸚鵡”、“光学幻觉”和“神话中的城市”这样的名字。）花 30 分钟设计一个滑稽的标识，并把它用在你的备忘录和报告上。在与别人交谈时，大方地使用你的团队的名字。这听起来很傻，但它能给你的团队一个用于建设的身份标识，并给世界以某种难忘的、可以与你们的工作相关联的东西。

不要重复你自己

在“重复的危害”（26 页）中，我们讨论了消除团队成员之间的重复工作的困难。这样的重复会造成工作的浪费，并且可能会带来维护的噩梦。显然，良好的交流可以有所帮助，但有时还需要另外的一些东西。

有些团队指定某个成员担任项目资料管理员，负责协调文档和代码仓库。其他团队成员在查找资料时，可以首先找这个人。通过阅读正在处理的材料，好的资料管理员还能发现正在逼近的重复。

当项目对一个资料管理员来说太大时（或是在无人愿意担任这一职务时），可以指定多人负责工作的各个方面。如果有人想要讨论日期处理，他们知道应该去找 Mary。如果有数据库 schema 问题，去找 Fred。

同时，不要忘了群件系统和本地 Usenet 新闻组在交流、以及问题和解答存档方面的价值。

⁴⁴ 团队用一个声音说话——对外部。在内部，我们强烈地鼓励进行活跃、热烈的辩论。好开发者往往对工作有很大的热情。

正交性

传统的团队组织基于老式的软件构造瀑布方法。各个个体的角色是基于其工作职责指派的。你会发现商业分析师、架构师、设计师、程序员、测试员、资料管理员，等等⁴⁵。这里有一个隐含的层级关系——按照对你的授权，你越接近用户，级别就越高。

有些开发文化把事情推向极端，实施严格的责任划分：编码员不许与测试员交谈，后者又无须与首席架构师交谈，等等。于是有些组织通过让不同的子团队沿不同的管理链进行报告，对问题进行隔离。

认为项目的各种活动——分析、设计、编码、测试——会孤立地发生，这是一个错误。它们不会孤立发生。它们是看待同一问题的不同方式，人为地分隔它们会带来许多麻烦。离代码的用户有两、三层远的程序员不大可能注意到他们的工作的应用语境。他们将无法做出有见识的决策。

提示 60

Organize Around Functionality, Not Job Functions
围绕功能、而不是工作职责进行组织

我们喜欢按照功能划分团队。把你的人划分成小团队，分别负责最终系统的特定方面的功能。让各团队按照各人的能力，在内部自行进行组织。每个团队都按照他们约定的承诺，对项目中的其他团队负有责任。承诺的确切内容随项目而变化，团队间的人员分配也是如此。

⁴⁵ 在 *The Rational Unified Process: An Introduction* 中，作者标识出项目团队中的 27 种不同角色！
[Kru98]

这里的功能并不必然意味着最终用户的用例。数据库访问层是功能，帮助子系统也是功能。我们是在寻求内聚的、在很大程度上自足的团队——和我们在使代码模块化时应该使用的标准完全一样。如果团队的组织是错误的，会有一些警告性迹象——一个经典的例子是有两个子团队在做同一个程序模块或类。

这样按功能组织有什么好处？使用我们用于组织代码的相同技术，去用像合约（按合约设计，109页）、解耦（解耦与得墨忒耳法则，138页）、正交性（正交性，34页）这样的技术组织我们的各种资源，有助于使团队作为整体与变化的各种效应隔离开来。如果用户突然决定变更数据库供应商，只有数据库团队应该受影响。如果市场部门突然决定使用现成工具来实现日程安排功能，只有日程小组会受影响。适当地加以运用，这种分组方式能够极大地减少各个开发者的工作之间的相互影响、缩短时间标度、提高质量、并减少缺陷的数目。这种途径还能带来更愿意付出的开发者。每个团队都知道他们自己要对特定的功能负责，所以他们更会觉得自己是他们的工作成果的主人。

但是，只有在项目拥有负责的开发者、以及强有力的项目管理时，这种途径才有效。创立一组自行其是的团队并放任自流，是一种灾难性的处方。项目至少需要两个“头”——一个主管技术，另一个主管行政。技术主管设定开发哲学和风格，给各团队指派责任，并仲裁成员之间不可避免的“讨论”。技术主管还要不断关注大图景，设法找出团队之间任何不必要的、可能降低总体正交性的交叉。行政主管（或项目经理）调度各团队所需的各种资源，监视并报告进展情况，并根据商业需要帮助确定各种优先级。在与外界交流时，行政主管还要充当团队的大使。

大型项目的团队需要额外的资源：负责索引和存储代码及文档的资料管理员，负责提供常用工具及环境、并进行运行支持的工具构建员，等等。

这种类型的团队组织在精神上与老式的主程序员团队概念（首见于1972年[Bak72]）类似。

自动化

确保一致和准确的一种很好的方式是使团队所做的每件事情自动化。如果你的编辑器能够自动在你输入时安排代码的布局，为什么要手工进行呢？如果夜间构建能够自动运行各种测试，为什么要手工完成测试表单呢？

自动化是每个项目团队的必要组成部分——重要得足以让我们从下一页开始，专用一节加以讨论。为了确保事情得以自动化，指定一个或多个团队成员担任工具构建员，构造和部署使项目中的苦差事自动化的工具。让它们制作 makefile、shell 脚本、编辑器模板、实用程序，等等。

知道何时停止绘画

要记住，团队是由个体组成的。让每个成员都能以他们自己的方式闪亮。给他们足够的空间，以支持他们，并确保项目的交付能够符合需求。然后，要像“足够好的软件”（11 页）中的画家一样，抵抗不断画下去的诱惑。

相关内容：

- 软件的熵，4 页
- 石头汤与煮青蛙，7 页
- 足够好的软件，9 页
- 交流！，18 页
- 重复的危害，34 页
- 按合约设计，109 页
- 解耦与得墨忒耳法则，138 页
- 无处不在的自动化，230 页

挑战

- 在软件开发领域以外找一找成功的团队。他们成功的原因是什么？他们是否使用了

这一节讨论的任何过程？

- 下一次启动项目时，设法说服别人为其创立品牌。给你的团队时间适应这个主意，然后快速地核查一下，看它在团队内部和外部都有什么作用
- 团队代数：在学校里，我们会遇到这样的问题：“如果4个工人挖一条沟需要6个小时，8个工人需要多少时间？”但是，在现实生活中，哪些因素会影响下面这个问题的答案：“4个程序员开发某个应用需要6个月，8个程序员需要多少时间？”在多少种情况下，所需时间实际上会减少？

42 无处不在的自动化

文明通过增加我们不加思索就能完成的重要操作的数目而取得进步。

——阿尔弗雷德·诺思·怀特海

在汽车时代的破晓时分，启动一辆T型福特车的操作说明有两页还不止。而驾驶现代汽车，你只需转动钥匙——启动过程是自动的，十分简单。遵循一串指令进行操作的人可能会撑掉引擎，而自动启动器却不会。

尽管计算行业仍处在T型福特车的阶段，我们无法承受为一些常用操作而反复阅读两页长的操作说明。无论是构建和发布流程、是书面的代码复查工作、还是其他任何在项目中反复出现的任务，都必须是自动的。我们也许必须在一开始就构建启动器和喷油器，但它们一旦完成，我们从此只需转动钥匙就可以了。

此外，我们想要确保项目的一致性和可重复性。人工流程不能保证一致性，也无法保证可重复性，特别是在不同的人对流程的各个方面有不同解释时。

一切都要自动化

在某个客户的开发现场，我们曾经看到所有开发者都在使用同一种 IDE。他们的系统管理员给了每个开发者一套说明，告诉他们怎样把附加软件包安装到 IDE 中。说明有许多页——到处都写着点击这里、滚动那里、拖这个、双击那个、以及再做一遍。

并不奇怪，每个开发者的机器里的内容有着轻微的不同。当不同的开发者运行相同的代码时，应用的行为会出现微妙的差异。bug 会在一台机器上出现，在其他机器上却不出现。追踪任何一个组件的版本差异通常都会揭示出一种让人意想不到的情况。

提示 61

Don't Use Manual Procedures

不要使用手工流程

人的可重复性并不像计算机那么好。我们也不应期望他们能那样。shell 脚本或批处理文件能以相同的次序、反复执行同样的指令。它们能被置于源码控制之下，你因而也可以检查流程的修改历史（“但它本来能工作……”）。

另一个特别受欢迎的自动化工具是 cron（或 Windows NT 上的“at”）。它允许我们安排无人照管的任务周期性地运行——通常是在午夜。例如，下面的 crontab 文件指定，每天午夜过后 5 分钟运行项目的 nightly 命令，每个工作日的凌晨 3:15 进行备份，每月第一天的午夜运行 expense_reports。

```
# MIN HOUR DAY MONTH DAYOFWEEK  COMMAND
# -----
5      0    *    *    *           /projects/Manhattan/bin/nightly
15     3    *    *    1-5         /usr/local/bin/backup
0      0    1    *    *           /home/accounting/expense_reports
```

使用 cron，我们可以自动安排备份、夜间构建、网站维护、以及其他任何可以无人照管地完成的事情。

项目编译

项目编译是一件应该可靠、可重复地进行的琐碎工作。我们通常通过 `makefile` 编译项目，即使是在使用 IDE 环境时。使用 `makefile` 有若干好处。它是脚本化、自动化的流程。我们可以增加挂钩，让其为我们生成代码，并自动运行回归测试。IDE 有自身的优势，但只用 IDE，可能很难获得我们寻求的自动化程度。我们想要用一条命令就完成签出、构建、测试和发布。

生成代码

在“重复的危害”（26 页）中，我们提倡生成代码，以根据公共来源派生知识。我们可以利用 `make` 的依赖分析机制，让这一过程变得更容易。给 `makefile` 增加规则，根据其他来源自动生成文件，是一件相当简单的事情。例如，假定我们想要根据一个 XML 文件生成一个 Java 文件，并编译所得结果。

```
.SUFFIXES: .Java .class .xml
.xml.java:
    perl convert.pl $< > $@
.Java.class:
    $(JAVAC) $(JAVAC_FLAGS) $<
```

敲入 `make test.class`，`make` 就会自动查找名为 `test.xml` 的文件，通过运行 Perl 脚本构建一个 `.java` 文件，然后编译该文件，产生 `test.class`。

我们还可以用同样的一组规则，根据其他形式的文件，自动生成源代码、头文件、或是文档（参见“代码生成器”，102 页）

回归测试

你还可以让 `makefile` 为你运行回归测试，或是针对单个模块，或是针对整个子系统。只要在源码树顶部发出一条命令，你就可以轻松地测试整个项目；也可在单个目录中发出同样的命令，测试单个模块。关于回归测试，详见“无情的测试”（237 页）。

递归 make

许多项目会为项目构建和测试设置递归的、层次结构的 makefile。但要注意一些潜在的问题。

make 会计算出它必须构建的各个目标之间的依赖关系。但它只能分析存在于单次 make 调用中的依赖关系。特别地，递归的 make 不知道对 make 的其他调用可能具有的依赖关系。如果你小心严谨，你可以得到正确的结果，但那很容易带来不必要的额外工作——或是遗漏某一依赖，在需要重新编译时没有重新编译。

此外，构建依赖可能会和测试依赖不一样，你也许需要不同的层次结构。

构建自动化

构建是这样一个过程：取一个空目录（和一个已知的编译环境），从头开始构建项目，产生所有你希望产生的、最终交付的东西——例如，CD-ROM 主映像或自解开的存档。在典型情况下，项目构建将包括以下几个步骤：

1. 从仓库中签出源码。
2. 从头开始构建项目，在典型情况下是从顶层的 makefile 开始。每次构建都会标注某种形式的发布或版本号，或是日期戳。
3. 创建可分发映像。这个过程可能需要确定文件所有权和权限，并严格按照发运时所需的格式⁴⁶，产生所有例子、文档、README 文件，以及将随同产品发运的任何东西。
4. 运行规定的测试（make test）。

⁴⁶ 例如，如果你是在制作 ISO9660 格式的 CD-ROM，你可以运行能产生 9660 文件系统的按位映射的映像的程序。为何要等到发运前夜才来确保它能工作呢？

对于大多数项目，这一层面的构建是在每天夜间自动运行的。在典型情况下，与在构建项目的某个具体部分时运行的测试相比，在这样的夜间构建中运行的测试要更完整。要点在于，要让完全构建（full build）运行所有可用的测试。你想要知道今天对代码做出的一处改动是否是某个回归测试失败的原因。通过在靠近问题源头的地方确定问题，你更有可能找到并修正它。

如果你没有定期运行测试，你可能会发现，应用因为 3 个月前做出的代码改动而失败。但愿你有足够的好运气找到它。

最终构建

你想要作为产品发运的最终构建，可能具有与常规的夜间构建不同的需求。最终构建可能要求锁住仓库，或是标上发布号；要求设置不同的优化和调试标志，等等。我们喜欢使用一个单独的 make 目标（比如 `make final`），一次完成所有这些参数设置。

要记住，如果产品的编译方式与先前的版本不同，你必须针对这个版本再次进行所有测试。

自动化管理

如果程序员能够把他们的所有时间投入实际编程，那不是很好吗？遗憾的是，难得有这样的情况。有 e-mail 要回复，有书面工作要完成，有文档要发布到 Web 上，等等。你可以决定创建一个 shell 脚本来完成一些烦人的工作，但你仍然要记得在需要时运行这个脚本。

因为记忆是随着你年龄的增长而丧失的第二种东西⁴⁷，我们不想过分依赖它。我们可以运行脚本，让它们基于源码和文档的内容，自动为我们完成各种流程。我们的目标是维持自动、无人照管、内容驱动的工作流。

⁴⁷ 第一种是什么？我忘了。

网站生成

许多开发团队用内部网站来进行项目交流，我们认为这是一个很好的想法。但我们不想花太多时间维护网站，也不想让它变得陈旧或过时。误导人的信息比完全没有信息还要糟糕。

从代码、需求分析、设计文档中提取的文档以及任何图片、图表或图形都需要定期发布到网站上。我们想要让这些文档作为夜间构建的一部分，或是作为源码签入流程中的一个挂钩，自动发布出去。

无论它是怎样完成的，Web 内容都应该根据仓库中的信息自动生成，并且无需人的干预就发布出去。这其实是 *DRY* 原则的另一种应用：信息以已签入代码和文档这一种形式存在。从 Web 浏览器的角度看出来的视图——只是视图。你应该不必手工维护该视图。

夜间构建生成的任何信息都应该能在开发网站上访问：构建自身的结果（例如，构建结果可以通过一页摘要的方式给出，其中包括编译警告、错误，以及当前状态），回归测试，性能统计，编码度量，以及任何其他的静态分析，等等。

批准流程

有些项目具有各种必须遵循的管理工作流。例如，需要安排代码或设计复查，需要批准，等等。我们可以使用自动化——特别是网站——帮助减轻书面工作负担。

假定你想要使代码复查安排和批准自动化，你可以在每个源文件里放置一个特殊标记：

```
/* Status: needs_review */
```

可以用一个简单的脚本检查所有的源码，并查找具有 `needs_review` 状态的所有文件——这表明它们已准备好接受复查。随后你可以把这些文件的列表作为网页发布出去，或是自动发送 e-mail 给适当人员，甚或是使用某种日程软件自动安排一次会议。

你可以在网页上设置一个表单，用于让复查者登记文件是否通过了复查。在复查之后，状态可自动变为 `reviewed`。是否与所有参与者一起进行检查取决于你。你仍然

可以自动完成书面工作（在 CACM 1999 年 4 月的一篇文章中，Robert Glass 总结了相关研究，这些研究似乎表明，尽管代码检查是有效的，通过会议进行复查却并非如此[Gla99a]）

鞋匠的孩子

鞋匠的孩子没鞋穿。软件开发人员常常会使用最糟糕的工具来完成工作。

但我们有制作更好的工具所需的所有原材料。我们有 cron。在 Windows 和 Unix 平台上，我们都有 make。我们还有 Perl 和其他一些高级脚本语言，可用于快速开发自制的工具、网页生成器、代码生成器、测试装备，等等。

让计算机去做重复、庸常的事情——它会做得比我们更好。我们有更重要、更困难的事情要做。

相关内容：

- 我的源码让猫给吃了，2 页
- 重复的危害，26 页
- 纯文本的力量，73 页
- shell 游戏，77 页
- 调试，90 页
- 代码生成器，102 页
- 注重实效的团队，224 页
- 无情的测试，237 页
- 全都是写，248 页

挑战

- 看一看你在工作日的习惯。你有没有发现任何重复的任务？你有没有一再敲入同样的命令序列？

试着写一些脚本，使该过程自动化。你有没有总是反复点击同样的图标序列？你能否创建宏，让它为你做所有这些事情？

- 你的项目的书面工作有多少可以自动化？假定编程人员的开销很高⁴⁸，确定项目预算有多少正浪费在管理过程上。基于其可能节省的总开销，你能否证明制作一个自动化解决方案所需花费的时间是合理的？

43 无情的测试

大多数开发者都讨厌测试。他们往往会温和地测试，下意识地知道代码会在哪里出问题，并避开那些薄弱的地方。**注重实效的程序员**与此不同。我们受到驱迫，现在就要找到我们的 bug，以免以后经受由别人找到我们的 bug 所带来的羞耻。

寻找 bug 有点像是用网捕鱼。我们用纤小的网（单元测试）捕捉小鱼，用粗大的网（集成测试）捕捉吃人的鲨鱼。有时鱼会设法逃跑，所以为了抓住在我们的项目池塘里游动的、越来越狡猾的缺陷，我们要补上我们发现的任何漏洞。

提示 62

Test Early. Test Often. Test Automatically.

早测试，常测试，自动测试。

一旦我们有了代码，我们就想开始进行测试。那些小鱼苗有飞快地变成吃人的大鲨鱼的可恶习惯，而抓住鲨鱼会困难许多。但我们又不想手工进行所有这些测试。

许多团队都会为他们的项目精心制订测试计划。有时他们甚至会使用这些计划。但我们发现，使用自动测试的团队成功的机会要大得多。与呆在架子上的测试计划相

⁴⁸ 为进行估算，你可以按照大约 100,000 美元/人的行业平均水平进行计算——其中包括薪水加奖金、培训、办公场所、以及杂项开销等费用。

比。随每次构建运行的测试要有效得多。

bug 被发现得越早，进行修补的成本就越低。“编一点，测一点”是 Smalltalk 世界里流行的一句话⁴⁹，我们可以把这句话当作我们自己的曼特罗，在编写产品代码的同时（甚至更早）编写测试代码。

事实上，好的项目拥有的测试代码可能比产品代码还要多。编写这些测试代码所花的时间是值得的。长远来看，它最后会便宜得多，而你实际上有希望制作出接近零缺陷的产品。

此外，知道你通过了测试将给你高度的自信：一段代码已经“完成”了。

提示 63

Coding Ain't Done 'Til All the Tests Run

要到通过全部测试，编码才算完成

你写出了一段代码，并不意味着你可以告诉你的老板或客户，说它已经完成。不是这样。首先，代码从不会真正完成。更重要的是，在它通过所有可用的测试之前，你不能声称它已经可供使用。

我们须要查看项目范围测试的三个主要方面：测试什么、怎样测试、以及何时测试。

测试什么

你需要进行的测试的主要类型有：

- 单元测试
- 集成测试
- 验证和校验（validation and verification）

⁴⁹ eXtreme Programming[URL 45]把这个概念称为“持续集成，无情测试”。

- 资源耗尽、错误及恢复
- 性能测试
- 可用性测试

这份列表绝不是完整的，有些特殊的项目还需要各种其他类型的测试，但它给了我们一个很好的出发点。

单元测试

单元测试是对某个模块进行演练的代码。我们在“易于测试的代码”（189页）中专门讨论过这一话题。单元测试是我们将在本节讨论的所有其他形式的测试的基础。如果各组成部分自身不能工作，它们结合在一起多半也不能工作。你使用的所有模块都必须通过它们自己的单元测试，然后你才能继续前进。

集成测试

集成测试说明组成项目的主要子系统能工作，并且能很好地协同。如果在适当的地方有好的合约，并且进行了良好的测试，我们就可以轻松地检测到任何集成问题。否则，集成就会变成肥沃的 bug 繁殖地。事实上，它常常是系统的 bug 来源中最大的一个。

集成测试实际上只是我们描述过的单元测试的一种扩展——只不过现在你在测试的是整个子系统遵守其合约的情况。

验证和校验

一旦你有了可执行的用户界面或原型，你需要回答一个最重要的问题：用户告诉了你他们需要什么，但那是他们需要的吗？

它满足系统的功能需求吗？这也需要进行测试。没有 bug，但回答的问题本身是错误的，这样的系统不会太有用。要注意用户的访问模式（access pattern）、以及这些

模式与开发者所用的测试数据的不同（例如，92页画笔的故事）

资源耗尽、错误及恢复

现在你已经很清楚，系统在理想条件下将会正确运行，你需要知道的是，它在现实世界的条件下将如何运行。在现实世界中，你的程序没有无限的资源；它们会把资源耗尽。你的代码可能遇到的一些限制包括：

- 内存空间
- 磁盘空间
- CPU 带宽
- 挂钟时间
- 磁盘带宽
- 网络带宽
- 调色板
- 视频分辨率

你可能会实际检查磁盘空间或内存分配的失败，但是否经常检查其他各项呢？你的应用适用于有 256 种颜色的 640 x 480 的屏幕吗？它能在有 24 位颜色的 1600 x 1280 的屏幕上运行，而不会看上去像一张邮票？批处理是否会在存档开始之前结束？

你可以检测环境的限制，比如视频参数，并进行相应的调整。但是，不是所有失败都是可以恢复的。如果你的代码检测到内存已经耗尽，你的选择有限：除了失败，你也许没有足够的资源去做任何事情。

当系统确实失败时⁵⁰，它会得体地失败吗？它会尽可能设法保存其状态、防止工作丢失吗？或是会当着用户的面造成“GPF”（General Protection Fault）或“core-dump”？

⁵⁰ 我们的文字编辑想让我们把这句话改成“如果系统确实失败……”，我们没有改。

性能测试

性能测试、压力测试或负载测试也可能是项目的一个重要方面。

问问你自己，软件是否能满足现实世界的条件下的性能需求——预期的用户数、连接数、或每秒事务数。它可以伸缩吗？

对于有些应用，你可能需要用专门的测试硬件或软件模拟现实情况下的负载。

可用性测试

可用性测试与到目前为止讨论过的其他测试类型不同。它是由真正的用户、在真实的环境条件下进行的。

根据人的因素考察可用性，需要处理需求分析过程中的任何误解吗？软件对于用户，就像是手的延伸吗？（我们不仅想让自己的工具顺手，也想让我们为用户创建的工具让他们觉得顺手。）

与验证与校验的情况一样，你需要尽早在还有时间更正时进行可用性测试。对于较大的项目，你可以引入人员因素（human factor）专家。（至少，玩一玩单向镜也很有意思。）

没能满足可用性标准就像是除零错误，是个大 bug

怎样测试

我们已经考察了要测试什么。现在我们将把注意力转向怎样测试，包括：

- 回归测试
- 测试数据
- 演练 GUI 系统
- 对测试进行测试
- 彻底测试

设计/方法学测试

你能否测试代码自身的设计和你用于构建软件的方法学？勉强说来，是的，你能。你通过分析各种度量——对代码的各个方面的测量——进行这样的测试。最简单的（常常也是最枯燥的）代码度量是代码行数——代码自身有多大？

你还可以用多种多样其他度量检查代码，包括：

- McCabe 圈复杂度度量（McCabe Cyclomatic Complexity Metric，测量决策结构的复杂度）
- 继承扇入（fan-in，基类数目）和扇出（fan-out，以这个模块为父模块的派生模块的数目）
- 响应集（参见“解耦与得墨忒耳法则”，138 页）
- 类耦合比（参见[URL 48]）

有些度量被设计为给出“合格”评定，而其他一些则只有进行比较才有用。就是说，你为系统中的每个模块计算这些度量，并对特定模块及其兄弟模块进行对比。在此通常会使用标准的统计技术（比如平均偏差与标准偏差）。

如果你发现某模块的度量与其他所有模块都显著不同，你需要问问自己，那是否正确。对于有些模块，“使曲线发生剧变”也许没问题。但如果没有好的理由，那可能就表示在模块中存在潜在的问题。

回归测试

回归测试把当前测试的输出与先前的（或已知的）值进行对比。我们可以确定我们今天对 bug 的修正没有破坏昨天可以工作的代码。这是一个重要的安全网，它能减少令人不快的意外。

我们到目前为止提到过的所有测试都可作为回归测试运行，确保我们在开发新代码时没有损失任何领地。我们可以运行回归测试，对性能、合约、有效性等进行校验。

测试数据

我们从何处获取运行所有这些数据所需的数据？数据只有两种：现实世界的数据和合成的数据。实际上我们两者都需要，因为这两类数据的不同性质将揭示出我们软件中的不同 bug。

现实世界的数据来自某种实际来源。它可能收集自己有系统、竞争者的系统、或是某种原型。它代表典型的用户数据。当你发现典型意味着什么时，你会大吃一惊。这最有可能揭示出需求分析中的缺陷和误解。

合成数据是人工生成的数据——或许受制于特定的统计约束。出于以下任何原因，你都有可能需要使用合成数据：

- 你需要大量数据，可能超过了任何现实世界的样本所能提供的数量。你也许可以用现实世界的数据做种子，生成更大的样本集，并且调整特定的有独特需要的字段。
- 你需要能强调边界条件的数据。这些数据可以完全由人工合成：含有 1999 年 2 月 29 日的日期字段、非常长的记录、或是带有外国邮政编码的地址。
- 你需要能展现出特定的统计属性的数据。想要看到如果每第三个事务都失败会发生什么？还记得在给予预先排好序的数据时会慢得像蜗牛的排序算法吗？你可以给出随机的或有序的数据，以暴露这种弱点。

演练 GUI 系统

测试 GUI 密集型系统常常需要专门的测试工具。这些工具可能基于简单的事件捕捉/重放模型，也可能需要用专门编写的脚本驱动 GUI。有些系统结合了这两者的基本要素。

不太完善的工具会在所测试的软件版本和测试脚本自身之间强加高度的耦合：如果你移动对话框，或是使按钮变小，测试也许就会找不到它，并且可能会失败。大多数现代的 GUI 测试工具都使用了一些不同的技术来绕开这一问题，并设法适应局部的

布局变化

但是，你无法使每一件事情都自动化。Andy 开发过一个图形系统，允许用户创建并显示模拟各种自然现象的、非确定性的可视效果。遗憾的是，在测试过程中，你不能简单地抓取一个位图，把输出与先前的运行的结果进行比较，因为它被设计成每一次都是不相同的。对于这样的情形，你也许别无选择，只能依赖于对测试结果进行人工解释。

编写解除了耦合的代码（参见“解耦与得墨忒尔法则”，138 页）的诸多优点之一是更加模块化的测试。例如，对于有 GUI 前端的数据处理应用，你的设计应该足够地解耦，以使你无需使用 GUI，就能对应用逻辑进行测试。这个主意与首先测试子组件类似。一旦应用逻辑得到了验证，就能更容易地定位在使用了用户界面的情况下出现的 bug（bug 很可能是用户界面代码制造的）。

对测试进行测试

因为我们不可能编写出完美的软件，所以我们也不可能编写出完美的测试软件。我们需要对测试进行测试。

把我们的测试套件集视为精心设计的保安系统，其用途是在出现 bug 时拉响警报。要测试保安系统，还有什么比设法闯入更好的办法呢？

在你编写了一个测试，用以检测特定的 bug 时，要故意引发 bug，并确定测试会发出提示。这可以确保测试在 bug 真的出现时抓住它。

提示 64

Use Saboteurs to Test Your Testing

通过“蓄意破坏”测试你的测试

如果你对测试真的很认真，你可以指定一个项目破坏员（project saboteur）。其职责是取源码树的一份单独的副本，故意引入 bug，并证实测试能抓住它们。

在编写测试时，确保警报在应该响起时响起。

彻底测试

一旦你自信你的测试是正确的，并且正在找出你制造的 bug，你怎么知道你已足够彻底地对代码库进行了测试呢？

简短的回答是“你不知道”，而且你也不会知道。但市场上有一些产品能够提供帮助。这些覆盖分析（coverage analysis）工具会在测试过程中监视你的代码，追踪哪些代码行执行过，哪些没有。这些工具能帮助你从总体上了解测试的全面程度，但别指望看到 100% 的覆盖率。

即使你真的碰巧测试了每一行代码，那也并非是整个的图景。重要的是你的程序可能具有的状态数。状态并非等同于代码行。例如，假定你有一个函数，用两个整数做参数，每个参数都可以是从 0 到 999 的数。

```
int test(int a, int b) {  
    return a < (a + b);  
}
```

在理论上，这个三行代码的函数有 1 000 000 种逻辑状态，其中 999 999 中能正确工作，1 种不能（当 a 和 b 都为 0 时）。只是知道你执行了这行代码并不会告诉你这一点——你需要确定程序所有可能的状态。遗憾的是，一般而言这真的是一个困难的问题。就像是“太阳会在你解决它之前变成又冷又硬的块”一样困难。

提示 65

Test State Coverage, Not Code Coverage

测试状态覆盖，而不是代码覆盖

即使具有良好的代码覆盖，你用于测试的数据仍然会有巨大的影响。而且，更为重要的是，你遍历代码的次序的影响可能是最大的。

何时进行测试

许多项目往往会把测试留到最后一分钟——最后期限（dead-line）马上就要来临时⁵¹。我们需要比这早得多地开始测试。任何产品代码一旦存在，就需要进行测试。

大多数测试都应该自动完成。注意到这一点很重要：我们所说的“自动”意味着对测试结果也进行自动解释。关于这一主题的更多讨论，参见“无处不在的自动化”（230页）。

我们喜欢尽可能频繁地进行测试，并且总是在我们把代码签入源码仓库之前。有些源码控制系统，比如 Aegis，可以自动完成这一工作。如果不能自动完成，我们就键入

```
%make test
```

通常，只要需要就回归地运行各个单元测试和集成测试，这并不成问题。

但有些测试可能不容易这样频繁地运行。例如，压力测试可能需要特殊的设置或设备，以及某种手工操作。这些测试的运行可以不那么频繁——也许每周或每月一次，但让它们按照计划定期运行，这很重要。如果它无法自动完成，那就确保让它出现在计划中，并给这项任务分配所有必需的资源。

把网收紧

最后，我们想要揭示一个最重要的测试概念。这是个显而易见的概念，而且事实上每本教科书都说要这样做。但出于某种原因，大多数项目仍然没有这样做。

如果有 bug 漏过了现有测试网，你需要增加新的测试，以在下一次抓住它。

⁵¹ **dead-line** \ded-lin\ n (1864) 在监狱里或监狱周围划出的一条线，犯人一旦越过，就有遭到枪击的危险。——韦伯斯特大学词典

提示 66**Find Bugs Once**

一个 bug 只抓一次

一旦测试人员找到了某个 bug，这应该是测试人员最后一次发现这个 bug。应该对自动化测试进行修改，从此每次都检查那个特定的 bug，没有例外，不管多琐碎，也不管开发者会怎样抱怨说：“哦，那决不会再发生了。”

因为它会再次发生。而我们完全没有时间去追踪自动化测试本可以为我们找到的 bug。我们必须把时间花在编写新的代码——以及新的 bug——上。

相关内容：

- 我的源码让猫给吃了，2 页
- 调试，90 页
- 解耦与得墨忒耳法则，138 页
- 重构，184 页
- 易于测试的代码，189 页
- 无处不在的自动化，230 页

挑战

- 你能否自动测试你的项目？许多团队都不得不回答“不能。”为什么？要定义可接受的结果太困难？这不会使你难以向出资人证明项目已经“完成”吗？
独立于 GUI 测试应用逻辑太困难吗？这说明 GUI 有什么问题？耦合？

44 全都是写

好记性不如烂笔头。

——中国谚语

在典型情况下，开发者不会太关注文档。在最好的情况下，它是一件倒霉的差事；在最坏的情况下，它就会被当作低优先级的任务，希望管理部门会在项目结束时忘掉它。

注重实效的程序员会把文档当作整个开发过程的完整组成部分加以接受。不进行重复劳动，不浪费时间，并且把文档放在手边——如果可能，就放在代码本身当中，文档的撰写就可以变得更容易。

这些并不完全是独创的或新奇的想法；把代码和文档紧密地结合在一起的思想早已出现：Donald Knuth 关于 literate programming（把像 TeX 这样的文本格式化语言和传统的编程语言结合在一起，使文档和源码保持在一起的编程方式——译注）的著作，在 Sun 公司的 JavaDoc 实用程序，还有其他一些地方。我们想要对代码和文档之间的分叉轻描淡写，而把它们当作同一模型的两个视图对待（参见“它只是视图”，157页）。事实上，我们想要再向前走一小步，既把我们的所有注重实效的原则应用于文档，也把它们应用于代码。

提示 67

Treat English as Just Another Programming Language

把英语当作又一种编程语言

为项目制作的文档基本上有两种：内部文档和外部文档。内部文档包括源码注释、设计与测试文档，等等。外部文档是发运或发布到外界的任何东西，比如用户手册。但不管目标受众是谁，也不管作者的角色是什么（开发者或技术文档撰写者），所有的文档都是代码的反映。如果有歧义，代码才最要紧——无论好坏。

提示 68

Build Documentation In, Don't Bolt It On

把文档建在里面，不要拴在外面

我们先来讨论内部文档。

代码中的注释

根据源码中的注释和声明产生格式化文档相当直截了当，但首先我们必须确保在代码中确实有注释。代码应该有注释，但太多的注释可能和太少一样糟。

一般而言，注释应该讨论为何要做某事、它的目的和目标。代码已经说明了它是怎样完成的，所以再为此加上注释是多余的——而且违反了 *DRY* 原则。

注释源码给你了完美的机会，让你去把项目的那些难以描述、容易忘记，却又不能记载在别的任何地方的东西记载下来：工程上的权衡、为何要做出某些决策、放弃了哪些替代方案，等等。

我们喜欢看到简单的模块级头注释、关于重要数据与类型声明的注释、以及给每个类和每个方法所加的简要头注释、用以描述函数的用法和任何不明了的事情。

当然，变量名应该精心选择，并且有意义。例如，`foo` 是没有意义的，`doit`、`manager` 或 `stuff` 也是如此。匈牙利表示法（即把变量的类型信息放在变量名自身里）在面向对象系统中绝对不合适。记住，你（以及其他的人）会好几百次地阅读代码，但却只编写几次。花时间拼写出 `connectionPool`，而不要只用 `cp`。

比无意义的名称更糟糕的是误导人的名称。你是否听到过有人像这样解释遗留代码中的不一致：“叫做 `getData` 的例程其实会把数据写到磁盘上”？人脑会反复混淆这样的情况——这叫做 **Stroop 效应** [Str35]。你可以自行尝试下面的试验，观察这种干扰的效应。取一些彩笔，用它们写下各种颜色的名称。但所写颜色名不要与所用彩笔的颜色相同。你可以把“蓝色”这个词写成绿色，把“棕色”这个词写成红色，等等。（另外的方法是到我们的网站 www.pragmaticprogrammer.com 上查看一组已经画好的颜色。）写好之后，试着尽可能快地大声说出你写下各个词所用的颜色。到一定时候你就会弄错，开始读出你写下的词，而不是它的颜色。名称对你的大脑具有深层的含义，而误导人的名称会增加你的代码的混乱。

你可以为参数建立文档，但问问你自己，这是否在所有情况下都真的有必要。JavaDoc 工具提倡的注释程度似乎是合适的：

```
/**
 * Find the peak (highest) value within a specified date
 * range of samples.
 *
 * @param aRange Range of dates to search for data.
 * @param aThreshold Minimum value to consider.
 * @return the value, or <code>null</code> if no value found
 *         greater than or equal to the threshold.
 *
 * public Sample findPeak(DateRange aRange, double aThreshold);
```

下面是不应出现在源码注释中的一些内容：

- **文件中的代码导出的函数的列表。**有些程序可以为你分析源码 使用它们，列表就保证是最新的
- **修订历史。**这是源码控制系统的用途所在（参见“源码控制”，86 页）但是，在注释中包括最后更改日期和更改人的信息可能是有用的⁵²
- **该文件使用的其他文件的列表。**使用自动工具可以更准确地确定这些信息
- **文件名。**如果在文件中必须出现文件名，不要手工进行维护。RCS 和类似的系统可以自动使这一信息保持最新 如果你移动文件或更改文件名，你不会希望必须记得编辑头注释

在源文件里应该出现的最重要的信息之一是作者的姓名——不一定是最后编辑文件的人，而是文件的所有者。使责任和义务与源码联系起来，能够奇迹般地使人保持诚实（参见“傲慢与偏见”，258 页）

⁵² 这类信息及文件名由 RCS \$Id\$ 标签提供

项目也许还要求在每个源文件中放置特定的版权提示或其他法律样本。让你的编辑器为你自动插入这些信息。

在源码中有了有意义的注释，像 JavaDoc[URL 7]和 DOC++[URL 21]这样的工具可以提取它们，并进行格式化，自动产生 API 级的文档。这是我们使用的一种更一般的技术——可执行文档——的一个特例。

可执行文档

假定我们有一个规范，要列出某个数据库表中的各个列，然后我们要用另外一组 SQL 命令在数据库中创建实际的表，还可能要创建某种编程语言的记录结构，用以存放表中一行的内容。同样的信息重复了三次。更改这三个信息源中的任何一个，其他两个就会立刻过时。这显然违反了 *DRY* 原则。

为了改正这一问题，我们需要选择权威的信息源。这可以是规范，是数据库 schema 工具，或者是第三种来源。让我们选择规范文档作为信息源。它现在是我们用于这一过程的模型（model）。然后我们需要找到一种途径，将其包含的信息作为不同的视图（view）导出——例如，数据库 schema 和高级语言记录⁵³。

如果你的文档是作为带有标记（markup）命令（例如，使用 HTML、LaTeX 或 troff）的纯文本存储的，那么你可以使用像 Perl 这样的工具自动提取 schema，并重新对其进行格式化。如果你的文档的格式是字处理器的二进制格式，那么就参考下一页方框里列举的其他选择。

你的文档现在是项目开发的完整组成部分。更改 schema 的唯一途径是更改文档。你可以保证，规范、schema 和代码全都是一致的。你使每次更改所必需的工作量减少到了最低程度，而且你可以自动更新所更改的各种视图。

⁵³ 更多关于模型与视图的讨论，参见“它只是视图”（157 页）

如果我的文档不是纯文本？

遗憾的是，现在越来越多的项目文档使用的字处理器会把文件以某种私有格式存储在磁盘上。我们说“遗憾”，是因为这严重地限制了你自动处理文档的选择。但是，你仍然有两种选择：

- **编写宏。**现在大多数完善的字处理器都有宏语言。通过一些努力，你可以用宏语言编程，把你的文档带有标签的部分导出为你所需的另外的格式。如果在这一层面编程太痛苦，你可以总是把适当的部分导出到标准格式的纯文本文件中，然后使用像 Perl 这样的语言将其转换为最终的格式。
- **使文档处于从属地位。**你可以把另外的表示、而不是文档当作决定性的信息源。（在数据库的例子中，你可以把 schema 当作权威信息源。）然后编写一个工具，把这些信息导出为文档可以导入的格式。但是要小心，你需要确保这些信息会在文档每次打印时、而不只是在文档创建时导入。

以一种类似的方式使用像 JavaDoc 和 DOC++ 这样的工具，我们可以根据源码生成 API 级的文档。模型是源码：模型的一种视图可以编译；其他视图意在用于打印或在 Web 上查看。我们的目标是总在模型上工作——不管模型是代码自身还是某种其他文档——并且让所有视图自动更新（更多关于自动化处理的讨论，参见“无处不在的自动化”，230 页）。

突然间，文档没有那么糟糕了。

技术文档撰写者

到现在为止，我们谈论的只是内部文档——由程序员自己撰写的文档。但如果在项目中有专业的技术文档撰写者又会怎样呢？情况常常是，程序员只是把材料“扔过隔板”，给那边的技术文档撰写者，并让他们自己设法制作用户手册、宣传材料，等等。

这是一个错误。程序员不撰写这些文档，并不意味着我们可以放弃注重实效的原则。我们想让文档撰写者和**注重实效的程序员**一样，接受同样的基本原则——特别是遵守 *DRY* 原则、正交性、模型—视图概念、以及自动化和脚本的使用。

打印还是编排

发布出的书面文档的一个固有问题是，它刚打印出来，可能就过时了。任何形式的文档都只是快照（snapshot）。

所以我们会设法把文档制作成能够在 Web 上在线发布的形式，用超链接整合在一起。使文档的这一视图保持最新，要比跟踪每一份已有的书面副本、烧掉它、重新打印并分发新副本更容易。这也是更好地满足广泛的听众的需求的途径。但要记住，要在每个网页上放上日期戳或版本号。这样，读者就可以很好地了解哪些内容是最新的，哪些是最近更改的，哪些没有更改过。

有很多时候，你需要以不同的格式给出同一份文档：打印的文档、网页、在线帮助、或是幻灯片。典型的解决方案极大地依赖于剪切—粘贴、根据原有文档创建一些新的独立文档。这是一种糟糕的做法：文档的表示形式应该独立于其内容。

如果你在使用标记（markup）系统，你就拥有按照你的需要实现任意多种不同输出格式的灵活性。你可以选择用

```
<H1>Chapter Title</H1>
```

在文档的报告版本中生成新的一章，给幻灯片版本中的一张新幻灯片加上标题、像 XSL 和 CSS⁵⁴这样的技术可用于根据这一标记生成多种输出格式。

⁵⁴ eXtensible Style Language 和 Cascading Style Sheets，两种为了帮助分离表示形式与内容而设计的技术。

如果你在使用字处理器，你很可能会拥有同样的能力。如果你记得用样式标识不同的文档元素，那么通过应用不同的样式表，你就可以彻底改变最终输出的外观。现在的大多数字处理器允许你把文档转换为各种格式，比如用于 Web 发布的 HTML。

标记语言

最后，对于大型文档项目，我们建议你考察一些更为现代的文档标记方案。

现在许多技术作者使用 DocBook 来定义他们的文档。DocBook 是一种基于 SGML 的标记语言，它仔细地标识了文档中的每一个组成成分。通过 DSSSL 处理器，文档可绘制成任意多种不同格式。Linux 文档项目使用了 DocBook 来以 RTF、TEX、info、PostScript 和 HTML 格式发布信息。

只要你原来的标记丰富得足以表达你所需的所有概念（包括超链接），转换成其他任何可发布形式可以很容易，也可以是自动化的。你可以制作在线帮助、印刷的手册、用于网站的产品指要、甚至是每日提示日历，全都依据同一来源——当然是在源码控制之下，并且与夜间构建一起构建（参见“无处不在的自动化”，230 页）。

文档和代码是同一底层模型的不同视图，但视图是惟一应该不同的东西。不要让文档变成二等公民，被排除在项目主要工作流之外。对待文档要像对待代码一样用心，用户（还有后来的维护者）会为你唱赞歌的。

相关内容：

- 重复的危害，26 页
- 正交性，34 页
- 纯文本的力量，73 页
- 源码控制，86 页
- 它只是视图，157 页
- 靠巧合编程，172 页
- 需求之坑，202 页
- 无处不在的自动化，230 页

挑战

- 你是否为你刚写的源码撰写了解释性的注释？为什么没有？有时间压力？不确定代码是否真的能工作——你只是在把某个想法当作原型进行试验？你以后要扔掉这些代码，对吗？这不会使它变成无注释、试验性的项目，是吗？
- 有时要为源码的设计建立文档并不让人舒服，因为设计在你的头脑中还不清晰；它仍然在演变。你觉得在它实际完成之前，你不应该浪费精力去描述它。这听起来像是靠巧合编程（172 页）？

45 极大的期望

诸天哪，要因此惊奇，极其恐慌……

——《耶利米书》2:12

某公司宣布利润创记录，其股价却下跌了 20%。当晚的金融新闻解释说，该公司没有实现分析家预期的业绩。一个小孩打开昂贵的圣诞礼物，却大哭起来——这不是他想要的廉价洋娃娃。某个项目团队奇迹般地实现了一个极其复杂的应用，但却遭到用户的抵制，因为该应用没有帮助系统。

在抽象的意义上，应用如果能正确实现其规范，就是成功的。遗憾的是，这只能付抽象的账。

在现实中，项目的成功是由它在多大程度上满足了用户的期望来衡量的。不符合用户预期的项目注定是失败的，不管交付的产品在绝对的意义上有多好。但是，像希望得到廉价洋娃娃的小孩的父母一样，你走得太远也会失败。

提示 69

Gently Exceed Your Users' Expectations

温和地超出用户的期望

但是，执行这条提示需要做一些工作。

交流期望

用户在一开始就会带着他们对所需要的东西的想象来到你面前。那可能不完整、不一致、或是在技术上不可能做到，但那是他们的，而且，就像过圣诞节的小孩一样，他们也在其中投入了一些感情。你不能简单地忽视它。

随着你对他们的需要的理解的发展，你会发现在他们的有些期望无法满足，或是他们的有些期望过于保守。你的部分角色就是要就此进行交流：与你的用户一同工作，以使他们正确地理解你将要交付的产品。并且要在整个开发过程中进行这样的交流。决不要忘了你的应用要解决的商业问题。

有些顾问称这一过程为“管理期望”（managing expectations）——主动控制用户对他们能从系统中得到什么应该抱有的希望。我们认为这是一个有点高人一等的想法。我们的角色不是控制用户的希望，而是要与他们一同工作，达成对开发过程和最终产品、以及他们尚未描述出来的期望的共同理解。如果团队能与外界通畅地交流，这个过程就几乎是自动的；每个人都应该理解所期望的是是什么以及它将被怎样构建出来。

有一些重要技术可用于促进这一过程。其中，“曳光弹”（48页）和“原型与便笺”（53页）是最为重要的技术。两者都让团队构造用户能看见的东西。两者都是与用户交流你对他们的需求的理解的理想途径。并且两者都让你和用户习惯于相互交流。

额外的一英里

如果你和用户紧密协作，分享他们的期望，并同他们交流你正在做的事情，那么当项目交付时，就不会发生多少让人吃惊的事情了。

这是一件**糟糕的事情**。要设法让你的用户惊讶。请注意，不是惊吓他们，而是要让他们高兴。

给他们的东西要比他们期望的多一点。给系统增加某种面向用户的特性所需的一点额外努力将一次又一次在商誉上带来回报。

随着项目的进展，听取你的用户的意见，了解什么特性会使他们真的高兴。你可以相对容易地增加，并让一般用户觉得很好的特性包括：

- 气球式帮助或工具提示帮助
- 快捷键
- 作为用户手册的补充材料的快速参考指南
- 彩色化
- 日志文件分析器
- 自动化安装
- 用于检查系统完整性的工具
- 运行系统的多个版本、以进行培训的能力
- 为他们的机构定制的 splash 屏幕（交互式软件显示的初始画面——译注）

所有这些特性都是相对表面的，而且实际上不会因为特性肿胀而给系统带来过度的负担。但是，每一项特性都告诉你的用户，开发团队想要开发出了不起的系统，要用于实际的系统。只是要记住，不要因为增加这些新特性而破坏系统。

相关内容：

- 足够好的软件，9 页
- 曳光弹，48 页
- 原型与便笺，53 页
- 需求之坑，202 页

挑战

- 有时，对项目最严厉的批评来自项目的开发者。你是否曾因你自己制作的东西未能满足自己的期望而感到失望？怎么会这样？也许这里有逻辑以外的东西在起作用。
- 当你交付软件时，你的用户有何评论？他们对应用各个方面的关注与你在其中投入的努力成比例吗？什么能使他们高兴？

46 傲慢与偏见

你愉悦我们已经足够长久。

——简·奥斯丁：《傲慢与偏见》

注重实效的程序员不会逃避责任。相反，我们乐于接受挑战，乐于使我们的专业知识广为人知。如果我们在负责一项设计，或是一段代码，我们是在做可以引以自豪的工作。

提示 70

Sign Your Work

在你的作品上签名

过去时代的手艺人为能在他们的作品上签名而自豪。你也应该如此。

但是，项目团队仍然是由人组成的，这条规则可能会带来麻烦。在有些项目里，代码所有权的概念可能会造成协作上的问题。人们可能会变得有“地盘”意识，或是不愿在公共的基础设施上工作。项目最后可能会变得像一些相互隔绝的小“采邑”。你变得怀有偏见，只欣赏自己的代码，排斥自己的同事。

那不是我们想要的。你不应该怀着猜忌心阻止要查看你的代码的人；出于同样的原因，你应该带着尊重对待他人的代码。“黄金法则”（“你要别人怎么对你，你就怎样对人”）和开发者间的相互尊重是使上面的提示行之有效的关键所在。

匿名（尤其是在大型项目中）可能会为邋遢、错误、懒惰和糟糕的代码提供繁殖地。只把自己看作齿轮上的一个齿、在无休无止的状况报告中制造蹩脚的借口，而不去编写优良的代码，那太容易了。

尽管代码必须有所有人，但其所有人不一定非得是个人。事实上，Kent Beck 的

成功的 eXtreme Programming 方法[URL 45]建议采用公共的代码所有权（但这还要求进行另外的实践，比如结对编程，以预防匿名的危险）。

我们想要看到对所有权的自豪。“这是我编写的，我对自己的工作负责。”你的签名应该被视为质量的保证。当人们在一段代码上看到你的名字时，应该期望它是可靠的、用心编写的、测试过的和有文档的，一个真正的专业作品，由真正的专业人员编写。

一个注重实效的程序员。

附录 A

资源

Resources

我们之所以能在本书中涵盖这么多内容，惟一原因是我们从高处俯瞰许多主题。如果我们对这些主题都给予应有的深入讨论，这本书的厚度就将是现在的十倍。

在本书的开头我们提出，**注重实效的程序员**应该不断学习。在这个附录中，我们列出了可以帮助你学习的一些资源。

在“专业协会”中，我们将详细介绍 IEEE 和 ACM。我们建议**注重实效的程序员**加入其中一个，或是两个都加入。随后，在“建设藏书库”中，我们重点列举了我们认为含有高质量的、恰切的信息（或者就是很有趣）的期刊、书籍和网站。

贯穿全书，我们引用了许多可以通过 Internet 访问的软件资源。在 Internet 资源中，我们列出了这些资源的 URL 以及对各个资源的简要描述。但是，Web 的本质意味着许多链接在你阅读本书时可能已经过时失效。你可以试着用搜索引擎查找更新的链接，或是访问我们的网站（www.pragmaticprogrammer.com），查看我们的链接区。

最后，本附录还附有本书的参考文献。

专业协会

有两个世界级的程序员专业协会：Association for Computing Machinery(ACM)⁵⁵和 IEEE Computer Society⁵⁶。我们建议所有程序员都加入其中一个，或是两个都加入。此外，美国以外的开发者可以加入本国的协会，比如英国的 BCS 协会。

成为专业协会的成员有许多好处。讨论会和本地的会议能让你有大量机会接触兴趣相投的人，而特别的兴趣团体和技术委员会能给予你机会、参与制订全世界使用的标准和指导方针。你还将从他们的出版物中学到许多知识，从高级的行业实践讨论直到低级的计算理论。

建设藏书库

我们喜爱阅读。如我们在“你的知识资产”（12 页）中所提到的，好的程序员总是在学习。不断阅读当前的书籍和期刊会对你有帮助。下面是我们喜欢的一些读物。

期刊

如果你和我们一样，你也会保存过期的杂志和期刊，直到它们堆得足够高，把底下的刊物压成扁平的“宝石”。这意味着，这些读物值得精心挑选。下面是我们阅读的一些期刊：

⁵⁵ ACM 会员服务：PO Box 11414, New York, NY 10286, USA. => www.acm.org

⁵⁶ 1730 Massachusetts Avenue NW, Washington, DC 20036-1992, USA. => www.computer.com

- **IEEE Computer** 寄送给 IEEE Computer Society 的会员 *IEEE Computer* 关注实践,但并不害怕理论。有些期围绕一个主题展开,而其他期就是有趣文章的汇总。这份杂志具有良好的信噪比。
- **IEEE Software** 这是 IEEE Computer Society 的另一份很好的双月刊,其目标读者是软件从业人员。
- **Communications of the ACM** ACM 的所有会员收到的一份基本的杂志。数十年来, *CACM* 一直是行业的标准,发表的开创性文章可能比其他任何来源都多。
- **SIGPLAN** 由 ACM Special Interest Group on Programming Languages 发行 *SIGPLAN* 是你作为 ACM 会员的一项附加选择。在上面常常发表语言规范、以及喜欢深入了解编程的人感兴趣的文章
- **Dr. Dobbs Journal** 月刊,可以订阅,也可以在报摊上购买。 *Dr. Dobbs* 有点怪异,但从位一级 (bit-level) 的实践到艰深的理论,文章的范围很广
- **The Perl Journal** 如果你喜欢 Perl,你很可能应该订阅 *The Perl Journal* (www.tpj.com)
- **Software Development Magazine** 关注项目管理和软件开发的一般问题的月刊

行业商务周刊

有几种为开发者及其经理发行的周报。这些报纸在很大程度上是一些公司发布的新闻、被重新编排为文章。但是,其内容仍然很有价值——它让你跟踪正在发生的事情、了解新产品的发布声明、并关注行业联盟的成形与瓦解。但不要指望在其中有大量技术内容

书籍

计算机书籍可能会很昂贵，但仔细加以选择，这是很值得的投入。下面是我们喜欢的一些书。

分析与设计

- ***Object-Oriented Software Construction, 2nd Edition***。Bertrand Meyer 的史诗般的著作，论述面向对象开发的基本原理，全书约 1,300 页[Mey97b]
- ***Design Patterns***。设计模式在比编程语言惯用手法更高的层面上描述解决特定类型的问题的途径。这本由 *Gang of Four* 撰写、现已成为经典的书籍[GHJV95]描述了 23 种基本的设计模式，包括 Proxy、Visitor 和 Singleton，等等
- ***Analysis Patterns***。一个高级架构型模式的宝藏，取自广泛的真实项目，“蒸馏”成书籍的形式。相对快速地深入了解多年建模经验的途径[Fow96]

团队与项目

- ***The Mythical Man Month***。Fred Brooks 的经典著作，论述项目团队组织的各种危险，最近作了修订[Bro95]。
- ***Dynamics of Software Development***。一系列论述大型团队软件构建的短文，着重讨论团队成员之间、团队与外界之间的动力机制[McC95]。
- ***Surviving Object-Oriented Projects: A Manager's Guide***。Alistair Cockburn 的“战地报导”，阐释 OO 项目管理的许多危险和陷阱——特别是你的第一个项目。Cockburn 先生提供了能让你解决最常见问题的提示与技术[Coc97b]。

具体环境

- ***Unix***。W. Richard Stevens 撰写了若干杰出的著作，包括 *Advanced Programming in the Unix Environment* 和 *Unix Network Programming*[Ste92, Ste98, Ste99]，等

等

- **Windows**。Marshall Brain 的 *Win32 System Services*[Bra95]是低级 API 的简明参考。Charles Petzold 的 *Programming Windows*[Pet98]是 Windows GUI 开发的权威书籍。
- **C++**。一旦你发现自己要参与 C++项目的开发，赶快跑（不要走）到书店去购买 Scott Meyer 的 *Effective C++*，可能还要 *More Effective C++*[Mey97a, Mey96]。要构建任何稍具规模的系统，你会需要 John Lakos 的 *Large-Scale C++ Software Design*[Lak96]。要了解各种高级技术，可向 Jim Coplien 的 *Advanced C++ Programming Styles and Idioms*[Cop92]求助。

此外，对于像 perl、yacc、sendmail、Windows 内幕、正则表达式这样的话题和语言，O'Reilly 的 Nutshell 系列（www.ora.com）给出了快捷、全面的论述。

Web

在 Web 上查找好内容很难。下面是一些我们一周至少查看一次的链接。

- **Slashdot**。宣称是“痴迷者的新闻，重要的资料”，Slashdot 是 Linux 社群的网络之家中的一个。除了定期更新 Linux 新闻之外，该网站还提供关于各种很棒的技术和影响开发者的问题的信息。
=> www.slashdot.org
- **Cetus Links**。关于面向对象话题的数千链接。
=> www.cetus-links.org
- **WikiWikiWeb**。Portland Pattern Repository 和模式讨论。这不仅是一个很好的网站，其“对各种想法进行集体编辑”的做法也是一项有趣的试验。
=> www.c2.com

Internet 资源

下面是 Internet 上的各种可用资源的链接。它们在本书撰写时是有效的，但（网络就是这样）当你阅读至此时，它们可能已经完全失效了。如果是这样，你可以试着用文件名进行大致的搜索，或是到**注重实效的程序员网站**（www.pragmaticprogrammer.com）查看我们给出的新链接。

编辑器

Emacs 和 vi 并非仅有的跨平台编辑器，但它们可自由获取，并且得到了广泛使用。快速浏览像 *Dr. Dobbs* 这样的杂志，你就可以找到几种另外的商业替代品。

Emacs

Emacs 和 XEmacs 都可以在 Unix 和 Windows 平台上使用。

[URL 1] The Emacs Editor

=> www.gnu.org

大型编辑器中的极品，含有任何编译器曾经有过的每一项特性。Emacs 有近乎垂直的学习曲线，但一旦你掌握了它，会有非常好的回报。它还有很好的邮件与新闻阅读器、地址簿、日程安排与日记、冒险游戏……

[URL 2] The XEmacs Editor

=> www.xemacs.org

几年前从原来的 Emacs 派生而出。XEmacs 被公认为内部更整洁、界面更漂亮。

vi

vi 有至少 15 种克隆，其中 vim 移植到的平台可能最多，所以如果你发现自己在许多不同的环境下工作，vim 是一个好选择。

[URL 3] The Vlm Editor

=> <ftp://ftp.fu-berlin.de/misc/editors/vim>

摘自文档：“对 vi 做了许多增强：多级撤消、多窗口与缓冲区、语法突显、命令行编辑、文件名完成、在线帮助、可视选择，等等……”

[URL 4] The elvis Editor

=> www.fh-wedel.de/elvis
支持 X 的增强型 vi 克隆。

[URL 5] Emacs Viper Mode

=> <http://www.cs.sunysb.edu/~kifer/emacs.html>
Viper 是一组使 Emacs 看起来像 vi 的宏。有人可能会问，为何要对世界上最大的编辑器进行扩展，让它模拟只有其部分能力的编译器。另外一些人则声称，Viper 结合了两者的最好部分。

编译器、语言和开发工具**[URL 6] GNU C/C++编译器**

=> www.fsf.org/software/gcc/gcc.html
世界上最流行的 C 和 C++编译器之一，也支持 Objective-C（在本书撰写时，先前从 gcc 分离出去的 egcs 项目正在合并回去）。

[URL 7] Sun 的 Java 语言

=> java.sun.com
Java 之家，含有可下载的 SDK、文档、教程、新闻，等等。

[URL 8] Perl 语言主页

=> www.perl.com
O'Reilly 是这组与 Perl 有关的资源的主人。

[URL 9] Python 语言

=> www.python.org
Python 是解释型、交互式的面向对象编程语言，语法有点奇特，拥有广泛而忠诚的追随者。

[URL 10] SmallEiffel

=> SmallEiffel.loria.fr
GNU Eiffel 编译器，能在任何有 ANSI C 编译器和 Posix runtime 环境的机器上运行。

[URL 11] ISE Eiffel

=> www.eiffel.com
Interactive Software Engineering 是“按合约设计”的发明者，销售商业 Eiffel 编译器及相关工具。

[URL 12] Sather

=> www.icsi.berkeley.edu/~sather

Sather 是一种从 Eiffel 发展而来的实验性语言。其目标是像 Common Lisp、CLU 或 Scheme 一样很好地支持更高阶的函数和迭代抽象,并像 C、C++ 或 Fortran 一样高效。

[URL 13] VisualWorks

=> www.objectshare.com

VisualWorks Smalltalk 环境之家。可以自由获取非商业的 Windows 和 Linux 版本。

[URL 14] Squeak 语言环境

=> squeak.cs.uiuc.edu

Squeak 是可自由获取、可移植的 Smalltalk-80 实现,用其自身编写而成;为改善性能,它可以产生 C 代码输出。

[URL 15] TOM 程序设计语言

=> www.gerbil.org/tom

一种源于 Objective-C 的非常动态的语言。

[URL 16] Beowulf 项目

=> www.beowulf.org

该项目通过廉价 Linux 机器的网络集群,构建高性能计算机。

[URL 17] iContract—Java 的按合约设计工具

=> www.reliable-systems.com

前条件、后条件与不变项的按合约设计形式体系,作为 Java 的预处理器实现。尊重继承,实现了存在量词 (existential quantifier), 等等

[URL 18] Nana—C 和 C++ 的日志及断言

=> <http://www.gnu.org/software/nana/nana.htm>

改善了 C 和 C++ 中的断言检查和日志记录支持。它还提供对按合约设计的一些支持。

[URL 19] DDD—数据显示调试器

=> www.cs.tu-bs.de/softech/ddd

可自由获取的 Unix 调试器图形前端

[URL 20] John Brant 的重构浏览器

=> st-www.cs.uiuc.edu/users/brant/Refactory

流行的 Smalltalk 重构浏览器。

[URL 21] DOC++文档生成器

=> www.zib.de/Visual/software/doc++/index.html

DOC++是用于 C/C++和 Java 的文档系统，能根据 C++头文件或 Java 类文件，直接生成 LaTeX 和 HTML 输出，以对你的文档进行高级的在线浏览。

[URL 22] xUnit-单元测试框架

=> www.XProgramming.com

一种简单而强大的概念。xUnit 单元测试框架为测试用各种语言编写的软件提供了致的平台

[URL 23] Tcl 语言

=> www.scriptics.com

Tcl (Tool Command Language) 是一种脚本语言，被设计为易于嵌入应用中。

[URL 24] Expect—与程序的自动交互

=> expect.nist.gov

在 Tcl[URL 23]之上构建的一种扩展 expect 允许你把与程序的交互编写成脚本。除了帮助你编写命令文件（例如，获取远程服务器上的文件，或是扩展你的 shell 的能力），expect 还可用于进行回归测试。图形版的 expectk 能让你用窗口前端包装非 GUI 应用。

[URL 25] T Spaces

=> www.almaden.ibm.com/cs/TSpaces

摘自其网页：“T Spaces 是一种具有数据库能力的网络通信缓冲区。它使由异种计算机和操作系统组成的网络中的应用与设备能够相互通信。T Spaces 提供群通信服务、数据库服务、基于 URL 的文件传送服务、以及事件通知服务。”

[URL 26] javaCC—Java 编译器-编译器

=> metamata.com

一种与 Java 语言紧密耦合在一起的解析器生成器

[URL 27] bison 解析器生成器

=> www.gnu.org/software/bison/bison.html

bison 读入文法规范，并据此生成适当解析器的 C 源码。

[URL 28] SWIG—简化的包装与接口生成器

=> www.swig.org

SWIG 是一种软件开发工具，能把用 C、C++ 和 Objective-C 编写的程序与各种高级编程语言（比如 Perl、Python 和 Tcl/Tk，以及 Java、Eiffel 和 Guile）连接在一起。

[URL 29] 对象管理组织（The Object Management Group, Inc.）

=> www.omg.org

OMG 是用于开发基于对象的分布式系统的各种规范的制订者。其工作范围包括 Common Object Request Broker Architecture (CORBA) 和 Internet Inter-ORB Protocol (IIOP)，等等。这些规范结合在一起，使对象能够相互通信，即使它们用不同语言编写，运行在不同类型的计算机上。

DOS 下的 Unix 工具

[URL 30] UWIN 开发工具

=> <http://www.gtllnc.com/uwin.htm>

Global Technologies, Inc., Old Bridge, NJ

UWIN 软件包提供的 Windows 动态链接库 (DLL) 能够模拟很大一部分 Unix C 一级的库接口。使用这一接口，GTL 已经把大量 Unix 命令行工具移植到 Windows 上。参见[URL 31]。

[URL 31] Cygnus 的 Cygwin 工具

=> sourceware.cygnus.com/cygwin/

Cygnus 软件包也对 Unix C 库接口进行了模拟，并在 Windows 操作系统下提供大量 Unix 命令行工具。

[URL 32] Perl Power Tools

=> www.perl.com/pub/language/pptr/

该项目用 Perl 重新实现经典的 Unix 命令集，使这些命令能在所有支持 Perl 的平台上运行（有很多这样的平台）。

源码控制工具

[URL 33] RCS—Revision Control System

=> prep.ai.mit.edu

GNU 的 Unix 和 Windows NT 源码控制系统。

[URL 34] CVS—Concurrent Version System

=> cvshome.org

可自由获取的 Unix 和 Windows NT 源码控制系统。通过支持客户 - 服务器模型和文件的并发访问，扩展了 RCS。

[URL 35] Aegis 基于事务的配置管理

=> <http://www.canb.auug.org.au/~millerp/aegis.html>

面向过程、推行项目标准的修订控制工具（比如校验签入的代码是否通过了测试）。

[URL 36] ClearCase

=> www.rational.com

版本控制、工作区与构建管理、过程控制。

[URL 37] MKS 源码完整性

=> www.mks.com

版本控制与配置管理。有些版本结合了一些特性，使远地开发者能在同一些文件上同时工作（很像 CVS）。

[URL 38] PVCS 配置管理

=> www.merant.com

源码控制系统，在 Windows 系统上非常流行。

[URL 39] Visual SourceSafe

=> www.microsoft.com

与 Microsoft 的可视开发工具集成在一起的版本控制系统。

[URL 40] Perforce

=> www.perforce.com

客户 - 服务器软件配置管理系统。

其他工具

[URL 41] WinZip—Windows 存档实用程序

=> www.winzip.com

Nico Mak Computing, Inc., Mansfield, CT

基于 Windows 的文件存档实用程序 既支持 zip 格式，也支持 tar 格式。

[URL 42] Z Shell

=> sunsite.auc.dk/zsh

一种为交互式用途设计的 shell，但同时也是强大的脚本语言，在 zsh 中结合了 bash、ksh 和 tcsh 的许多有用特性；另外还增加了许多新特性。

[URL 43] 可自由获取的 Unix 系统 SMB 客户

=> samba.anu.edu.au/pub/samba/

Samba 能让你在 Unix 和 Windows 系统间共享文件和其他资源。Samba 包括：

- SMB 服务器，给 SMB 客户（比如 Windows 95、Warp Server、smbfs，等等）提供 Windows NT 和 LAN Manager 风格的文件与打印服务。
- Netbios 名字服务器，提供浏览等支持。如果你愿意，Samba 可以成为你的 LAN 上的主浏览器。
- 类似 ftp 的 SMB 客户，允许你从 Unix、Netware 和其他操作系统访问 PC 资源（磁盘与打印机）。

论文与出版物

[URL 44] comp.object FAQ (Frequently Asked Questions)

=> www.cyberdyne-object-sys.com/oofaq2

comp.object 新闻组的内容充实、组织得当的 FAQ。

[URL 45] eXtreme Programming

=> www.XProgramming.com

摘自网站：“在 XP 中，我们使用量级非常轻的实践组合来创建团队，这样的团队能够迅速开发出极其可靠、高效、划分良好的软件。许多 XP 实践是作为 Chrysler C3 项目的组成部分而创建和测试的，该项目是用 Smalltalk 实现的非常成功的薪酬系统。”

[URL 46] Alistair Cockburn 的主页

=> members.aol.com/acockburn

查找“带着目标构造用例”和用例模板。

[URL 47] Martin Fowler 的主页

=> ourworld.CompuServe.com/homepages/martin_fowler

Analysis Patterns 的作者、*UML Distilled* 和 *Refactoring: Improving the Design of Existing Code* 的合著者。Martin Fowler 的主页讨论他的著作及他在 UML 方面的工作。

[URL 48] Robert C. Martin 的主页

=> www.objectmentor.com

关于面向对象技术的介绍性的好论文，这些技术包括依赖分析和度量（metrics）。

[URL 49] Aspect-Oriented（面向方面）编程

=> www.parc.xerox.com/csl/projects/aop/

以正交和声明的方式，给代码增加功能的一种途径。

[URL 50] JavaSpaces 规范

=> java.sun.com/products/javaspaces

用于 Java 的类 Linda 系统，支持分布式持久及各种分布式算法。

[URL 51] Netscape 源码

=> www.mozilla.org

Netscape 浏览器的开发源码。

[URL 52] 行话文件

=> www.jargon.org

Eric S. Raymond

许多常见的（和不那么常见的）计算机行业术语的定义，还有好些“民间传说”。

[URL 53] Eric S. Raymond 的论文

=> www.tuxedo.org/~esr

Eric 的论文 *The Cathedral and the Bazaar* 及 *Homesteading the Noosphere* 描述了开放源码运动的社会心理基础及影响。

[URL 54] K Desktop Environment

=> www.kde.org

摘自其网页：“KDE 是用于 Unix 工作站的功能强大的图形桌面环境。KDE 是一个 Internet 项目，在各个方面都是真正开放的。”

[URL 55] GNU Image Manipulation Program

=> www.gimp.org

Gimp 是自由分发的、用于图像创建、组合及修饰的程序。

[URL 56] 得墨忒耳项目

=> www.ccs.neu.edu/research/demeter

该研究关注的是采用 Adaptive Programming（自适应程序设计）使软件更易维护与演化。

其他

[URL 57] GNU 项目

=> www.gnu.org

Free Software Foundation, Boston, MA

自由软件基金会是为 GUN 项目筹措资金的免税组织。GUN 项目的目标是开发完整、自由的类 Unix 系统。他们开发的许多工具都已成为行业标准。

参考文献

- [Bak72] F.T.Baker. Chief programmer team management of production programming. *IBM Systems Journal*, 11(1):56-73, 1972.
- [BBM96] V.Basili, L.Briand, and W.L.Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751-761, October 1996.
- [Ber96] Albert J.Bernstein, *Dinosaur Brains: Dealing with All Those Impossible People at Work*. Ballantine Books, New York, NY, 1996.
- [Bra95] Marshall Brain. *Win32 System Services*. Prentice Hall, Englewood Cliffs, NJ, 1995.
- [Bro95] Frederick P.Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison Wesley, Reading, MA, anniversary edition, 1955.
- [CG90] N. Carriero and D.Gelerter, *How to Write Parallel Programs: A First Course*. MIT Press, Cambridge, MA, 1999.
- [CN91] Brad J. Cox and Andrex J. Novobilski, *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley, Reading, MA, 1991.
- [Coc97a] Alistair Cockburn, Goals and use cases, *Journal of Object Oriented Programming*, 9(7):35-40, September 1972.
- [Coc97b] Alistair Cockburn, *Surviving Object-Oriented Projects: A Manager's Guide*, Addison Wesley Longman, Reading, MA, 1997.
- [Cop92] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison Wesley, Reading, MA, 1992.
- [DL99] Tom Demarco and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House, New York, NY, second edition, 1999.

- [FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.
- [FOW96] Martin Fowler. *Analysis Pattern: Reusable Object Models*. Addison Wesley Longman, Reading, MA, 1999.
- [FS97] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison Wesley Longman, Reading, MA, 1997.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
- [Gla99a] Robert L. Glass. Inspections--Some surprising findings, *Communications of the ACM*, 42(4):17-19, April 1999.
- [Gla99b] Robert L. Glass. The realities of software technology payoffs. *Communications of the ACM*, 42(2):74-79, February 1999.
- [Hol78] Michael Holt. *Math Puzzles and Games*. Dorset Press, New York, NY, 1978.
- [Jac94] Ivar Jacobson. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison Wesley, Reading, MA, 1994.
- [KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241. Springer-Verlag, June 1997.
- [Knu97a] Donald Ervin Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison Wesley Longman, Reading, MA, third edition, 1997.
- [Knu97b] Donald Ervin Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley Longman, Reading, MA, third edition, 1997.
- [Knu98] Donald Ervin Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley Longman, Reading, MA, second edition, 1998.

- [KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison Wesley Longman, Reading, MA, 1999.
- [Kru98] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley Longman, Reading, MA, 1998.
- [Lak96] John Lakos. *Large-Scale C++ Software Design*. Addison Wesley Longman, Reading, MA, 1996.
- [LH89] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, Pages 38-48, September 1989.
- [Lis88] Barbara Liskov. Data abstraction and hierarchy, *SIGPLAN Notices*, 23(5), May 1988.
- [LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex and Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, Second edition, 1992.
- [McC95] Jim McCarthy. *Dynamics of Software Development*. Microsoft Press, Redmond, WA, 1995.
- [Mey96] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison Wesley Reading, MA, 1996.
- [Mey97a] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison Wesley Longman Reading, MA, second edition 1997.
- [Mey97b] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.
- [Pet98] Charles Petzold. *Programming Windows, The Definitive Guide to the Win32 API*. Microsoft Press, Redmond, WA, fifth edition, 1998.
- [Sch95] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, NY, second edition, 1995.
- [Sed83] Robert Sedgewick. *Algorithms*. Addison Wesley Reading, MA, 1983.

- [Sed92] Robert Sedgewick. *Algorithms in C++*. Addison Wesley Reading, MA, 1992.
- [SF96] Robert Sedgewick and Phillipe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley Reading, MA, 1996.
- [Ste92] W. Richard Stevens. *Advanced Programming in the Unix Environment*. Addison Wesley Reading, MA, 1992.
- [Ste98] W. Richard Stevens. *Unix Network Programming, Volume 1: Networking APIs: Sockets and Xti*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1998.
- [Ste99] W. Richard Stevens. *Unix Network Programming, Volume 2: Interprocess Communications*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1999.
- [Str35] James Ridley Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 18:643-662, 1935.
- [WK82] James Q. Wilson and George Kelling. The police and neighborhood safety. *The Atlantic Monthly*, 249(3): 29-38, March 1982.
- [YC86] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1986.
- [You95] Edward Yourdon. When good-enough software is best. *IEEE Software*, May 1995.

附录 B

练习解答

Answers To Exercises

练习 1：来自“正交性”（43 页）

你在编写一个叫做 `Split` 的类，其用途是把输入行拆分为字段。下面的两个 Java 类型构（signature）中，哪一个是更为正交的设计？

```
class Split1 {
    public Split1(InputStreamReader rdr) { ...
    public void readNextLine() throws IOException { ...
    public int numFields() { ...
    public String getField(int fieldNo) { ...
}
class Split2 {
    public Split2(String line) { ...
    public int numFields() { ...
    public String getField(int fieldNo) { ...
}
```

解答 1：我们的看法是，`Split2` 类更为正交。它专注于自己的任务，拆分输入行，同时忽略像输入行来自何处这样的细节。这不仅使代码更易于开发，也使得代码更为灵活。`Split2` 拆分的行可以来自文件、可以由另外的例程生成、也可以通过环境传入。

练习 2：来自“正交性”（43 页）

非模态对话框或模态对话框，哪一个能带来更为正交的设计？

解答 2：如果设计正确，很可能是非模态对话框。使用非模态对话框的系统与任一特定时刻正在发生的事情的关联较少。它很可能会拥有比模态系统更好的模块间通信基础设施，模态系统对系统的状态可能会有内在的假定——导致耦合增加、正交性降低

的假定

练习 3：来自“正交性”（43 页）

过程语言与对象技术的情况又如何？哪一种能产生更为正交的系统？

解答 3：这个问题有点棘手。对象技术可以提供更为正交的系统，但因为它有更多的特性可被滥用，使用对象技术实际上比使用过程语言更容易创建出非正交的系统。像多重继承、异常、操作符重载、父方法重定义（通过定义子类）这样的特性提供了通过不明显的途径增加耦合的充足机会。

练习 4：来自“原型与便笺”（56 页）

市场部门想要坐下来和你一起讨论一些网页的设计问题。他们想到用可点击的图像进行页面导航，但却不知道该用什么图像模型——也许是轿车、电话、或是房子。你有一些目标网页和内容；他们想要看到一些原型。哦，随便说一下，你只有 15 分钟。你可以采用什么样的工具？

解答 4：低技术可以搭救你！在白板上用笔画一些卡通——轿车、电话、房子——它们不必是伟大的艺术；用线条表示轮廓就很好。把描述目标页面内容的便笺放在可点击区域上。随着会议的进行，你可以改进图画和便笺的位置。

练习 5：来自“领域语言”（63 页）

我们想实现一种小型语言，用于控制一种简单的绘图包（也许是一种“海龟图形”（turtle-graphics）系统）。这种语言由单字母命令组成。有些命令后跟单个数字。例如，下面的输入将会绘制出一个矩形：

```
P 2 # select pen 2
D   # pen down
W 2 # draw west 2cm
N 1 # then north 1
E 2 # then east 2
S 1 # then back south
U   # pen up
```

请实现解析这种语言的代码。它应该被设计成能简单地增加新命令

解答 5: 因为我们想要使该语言可扩展, 我们将以表驱动的方式实现解析器。表中的每一项都含有命令字母、表明是否需要参数的标志、以及处理该命令要调用的例程的名称

```
typedef struct {
    char cmd;           * the command letter */
    int hasArg;          * does it take an argument */
    void (*func)(int, int); * routine to call */
} Command;
static Command cmds[] = {
    { 'P', ARG, doSelectPen },
    { 'U', NO_ARG, doPenUp },
    { 'D', NO_ARG, doPenDown },
    { 'N', ARG, doPenDir },
    { 'E', ARG, doPenDir },
    { 'S', ARG, doPenDir },
    { 'W', ARG, doPenDir }
};
```

主程序相当简单: 读取一行输入, 查找命令, 如果需要就获取参数, 然后调用处理器函数

```
while (tgets (buff, sizeof(buff), stdin)) {
    Command *cmd = findCommand(*buff);
    if (cmd) {
        int arg = 0;
        if (cmd->hasArg && !getArg(buff+1, &arg)) {
            fprintf(stderr, "'%c' needs an argument\n", *buff);
            continue;
        }
        cmd->func(*buff, arg);
    }
}
```

查找命令的函数对表进行线性搜索, 返回匹配项或 NULL

```
Command *findCommand(int cmd) {
    int i;
    for (i = 0; i < ARRAY_SIZE(cmds); i++) {
        if (cmds[i].cmd == cmd)
            return cmds + i;
    }
    fprintf(stderr, "Unknown command '%c'\n", cmd);
    return 0;
}
```

最后, 使用 sscanf 读取数值参数相当简单

```
int getArg(const char *buff, int *result) {
    return sscanf(buff, "%d", result) == 1;
}
```


练习 6: 来自“领域语言”(63 页)

设计一种解析时间规范的 BNF 文法 应能接受下面的所有例子:

4pm, 7:38pm, 23:42, 3:16, 3:16am

解答 6: 使用 BNF, 可以这样编写时间规范:

```

<time> ::= <hour> <amp;pm> |
          <hour> : <minute> <ompm> |
          <hour> : <minute>

<amp;pm> ::= am | pm
<hour>   ::= <digit> |
          <digit> <digit>

<minute> ::= <digit> <digit>

<digit>  ::= 0|1|2|3|4|5|6|7|8|9

```

练习 7: 来自“领域语言”(63 页)

用 yacc、bison 或类似的解析器生成器为练习 6 中的 BNF 文法实现解析器

解答 7: 我们使用 bison (yacc 的 GNU 版本) 编写我们的例子。为清晰起见, 我们在这里只给出解析器的主体, 全部实现见我们的网站上的源码。

```

time:      spec EOF
          { if ($1 >= 24*60) yyerror("Time is too large");
            printf("%d minutes past midnight\n", $1);
            exit(0);
          }
;

spec:      hour ':' minute
          { $$ = $1 + $3;
          }
        | hour ':' minute ampm
          { if ($1 > 11*60) yyerror("Hour out of range");
            $$ = $1 + $3 + $4;
          }
        | hour ampm
          { if ($1 > 11*60) yyerror("Hour out of range");
            $$ = $1 + $2;
          }
;

hour:      hour_num
          { if ($1 > 23) yyerror("Hour out of range");
            $$ = $1 * 60;
          }
;

```

```

minute:  DIGIT DIGIT
        { $$ = $1*10 + $2;
          if ($$ > 59) yyerror( "minute out of range" );
        };

ampm:    AM           { $$ = AM_MINS; }
        | PM          { $$ = PM_MINS; }
        ;

hour_num: DIGIT       { $$ = $1; }
        DIGIT DIGIT   { $$ = $1*10 + $2; }
        ;

```

练习 8: 来自“领域语言”(63页)

用 Perl 实现时间解析器(提示: 正则表达式可带来好的解析器)

解答 8:

```

$_ = shift;
/^(\\d\\d?)(am|pm)$/      && doTime ($1, 0, $2, 12);
/^(\\d\\d?):(\\d\\d)(am|pm)$/ && doTime($1, $2, $3, 12);
/^(\\d\\d?):(\\d\\d)$/      && doTime($1, $2, 0, 24);
die "Invalid time $_\\n";

#
# doTime(hour, min, ampm, maxHour)
#
sub doTime($$$$){
    my ($hour, $min, $offset, $maxHour) = @_;
    die "Invalid hour: $hour" if ($hour >= $maxHour);
    $hour += 12 if ($offset eq "pm");
    print $hour*60 + $min, " minutes past midnight\\n";
    exit(0);
}

```

练习 9: 来自“估算”(69页)

有人问你:“1Mbps 的通信线路和在口袋里装了 4GB 磁带、在两台计算机间步行的人, 哪一个的带宽更高?” 你要对你的答案附加什么约束, 以确保你的答复的范围是正确的?(例如, 你可以说, 访问磁带所花时间忽略不计)

解答 9: 我们的解答必须以一些假定为前提:

- 磁带中含有我们需要传送的信息。
- 我们知道人行走的速度。

- 我们知道两台机器间的距离
- 我们不考虑把信息转入和转出磁带所用的时间
- 在磁带上存储数据的开销大致与通过通信线路发送数据的开销相等

练习 10：来自“估算”（69 页）

那么，哪一个带宽更高？

解答 10：按照解答 9 中的说明：4GB 的磁带含有 32×10^9 比特信息，于是在 1Mbps 的线路上传送相同数量的信息需要约 32,000 秒，也就是约 9 小时。如果人以 $3\frac{1}{2}$ 英里的时速行走，那么两台机器至少需要相距 31 英里，通信线路才会胜过我们的传信人；否则，人将胜出。

练习 11：来自“文本操纵”（102 页）

你的 C 程序使用枚举类型表示 100 种状态。为进行调试，你想要能把状态打印成（与数字对应的）字符串。编写一个脚本，从标准输入读取含有以下内容的文件：

```
name
state_a
state_b
:
:
```

生成文件 *name.h*，其中含有：

```
extern const char* NAME_names[];
typedef enum {
    state_a,
    state_b,
    :
} NAME;
```

以及文件 *name.c*，其中含有：

```
const char* NAME_names[] = {
    "state_a",
    "state_b",
    :
};
```

解答 11：我们用 Perl 实现我们的解答

```

my @consts;
my $name = <>;
die "Invalid format - missing name" unless defined($name);
chomp $name;
# Read in the rest of the file
while (<>) {
    chomp;
    s/^\s*//; s/\s*$//;
    die "Invalid line: $_" unless /^(\w+)\s/;

    push @consts, $_;
}

# Now generate the file
open(HDR, ">$name.h") or die "Can't open $name.h: $!";
open(SRC, ">$name.c") or die "Can't open $name.c: $! ";

my $uc_name = uc($name);
my $array_name = $uc_name . "_names";

print HDR "/* file generated automatically - do not edit */\n";
print HDR "extern const char *$ {uc_name}_name[];\n";
print HDR "typedef enum {\n ";
print HDR join ",\n ", @consts;
print HDR "\n} $uc_name;\n\n";

print SRC "/* File generated automatically do not edit */\n";
print SRC "const char *$ {uc_name}_name[] = {\n \"";
print SRC join "\",\n \"", @consts;
print SRC "\"\n};\n";

close(SRC);
close(HDR);

```

按照 *DRY* 原则，我们不会把这个新文件剪贴进我们的代码中。相反，我们将用 `#include` 包含它——平板文件是这些常量的主来源。这意味着，我们需要一个 `makefile`，在文件发生变化时重新生成头文件。下面的内容片段摘自我们源码树中的测试床（可在网站上获取）。

```

etest.c etest.h: etest.inc enumerated.pl
                perl enumerated.pl etest.inc

```

练习 12：来自“文本操纵”（102 页）

在本书撰写中途，我们意识到我们没有把 `use strict` 指示放进我们的许多 Perl 例子中。编写一个脚本，检查某个目录中的 `.pl` 文件，给没有 `use strict` 指示的所有文件在初始注释块的末尾加上该指示。要记住给你改动的所有文件保留备份。

解答 12: 下面是我们的解答，用 Perl 编写

```
my $dir = shift or die "Missing directory";
for my $file (glob("$dir/*.pl")) {
    open(IP, "$file") or die "Opening $file: $! ";
    undef $/; # Turn off input record separator --
    my $content = <IP>; # read whole file as one string.
    close(IP);
    if ($content !~ /^use strict/m) {
        rename $file, "$file.bak" or die "Renaming $file: $! ";
        open(OP, ">$file") or die "Creating $file: $! ";
        # Put 'use strict' on first line that
        # doesn't start '#'
        $content = "s/^(?!#)/\nuse strict;\n\n/m;
        print OP $content;
        close(OP);
        print "Updated $file\n";
    }
    else {
        print "$file already strict\n";
    }
}
```

练习 13: 来自“代码生成器”(106 页)

编写一个代码生成器，读取图 3.4 中的输入文件，以你选择的两种语言生成输出。设法使其易于增加新语言。

解答 13: 我们用 Perl 实现我们的解决方案。它动态地加载模块来生成所需语言，所以增加新语言很容易。主例程（基于命令行参数）加载后端，然后读取其输入，基于每行的内容调用代码生成例程。我们没有过分担心出错处理——如果出了问题，我们很快就会知道

```
my $lang = shift or die "Missing language";
$lang .= "_cg.pm";
require "$lang" or die "Couldn't load $lang";
# Read and parse the file
my $name;
while (<>) {
    chomp;
    if (/^\s*S/) { CG::blankLine(); }
    elsif (/^\#(.*)/) { CG::comment($1); }
    elsif (/^M\s*(.+)/) { CG::startMsg($1); $name = $1; }
    elsif (/^E/) { CG::endMsg($name); }
    elsif (/^P\s*(\w+)$/) { CG::simpleType($1,$2); }
```

```

    elsif (/^F\s*(\w+)\s+(\w+)\s+(\d+)\s$/)
        { CG::arrayType($1,$2,$3); }
    else {
        die "Invalid line: $_";
    }
}

```

编写语言后端很简单：提供一个模块，实现所需的六个进入点。下面是 C 生成器：

```

#!/usr/bin/perl -w
package CG;
use strict;
# Code generator for 'C' (see cg_base.pl)
sub blankLine() { print "\n"; }
sub comment()   { print "/*$_[0] */\n"; }
sub startMsg()  { print "typedef struct {\n"; }
sub endMsg()    { print "} $_[0];\n\n"; }
sub arrayType() {
    my ($name, $type, $size) = @_;
    print " $type $name\[$size\];\n";
}
sub simpleType() {
    my ($name, $type) = @_;
    print " $type $name;\n";
}
1;

```

下面是 Pascal 生成器：

```

#!/usr/bin/perl -w
package CG;
use strict;
# Code generator for 'Pascal' (see cg_base.pl)
sub blankLine() { print "\n"; }
sub comment()   { print "{$_[0]} \n"; }
sub startMsg()  { print "$_[0] = packed record\n"; }
sub endMsg()    { print "end;\n\n"; }
sub arrayType() {
    my ($name, $type, $size) = @_;
    $size--;
    print " $name: array[0..$size] of $type;\n";
}
sub simpleType() {
    my ($name, $type) = @_;
    print " $name: $type;\n";
}
1;

```

练习 14: 来自“按合约设计”(118 页)

好合约有什么特征? 任何人都可以增加前条件和后条件, 但那是否会给你带来任何好处? 更糟糕的是, 它们实际上带来的坏处是否会大过好处? 对于下面的以及练习 15 和 16 中的例子, 确定所规定的合约是好, 是坏, 还是很糟糕, 并解释为什么

首先, 让我们看一个 Eiffel 例子。我们有一个用于把 STRING 添加到双向链接的循环链表中的例程 (别忘了前条件用 `require` 标注, 后条件用 `ensure` 标注)

```
-- Add an item to a doubly linked list,
-- and return the newly created NODE.
add_item (item : STRING) : NODE is
  require
    item /= Void -- '/' is 'not equal'.
  deferred -- Abstract base class.
  ensure
    result.next.previous = result -- Check the newly
    result.previous.next = result -- added node's links.
    find_item(item) = result -- Should find it.
  End
```

解答 14: 这个 Eiffel 例子是好合约。我们要求传入的数据不为 null, 同时我们保证双向循环链表的语义得到遵守。它还能帮助我们找到我们存储的字符串。因为这是 `deferred` 类, 实际实现它的类可以自由使用它想要使用的无论什么底层机制。它可以选择使用指针、数组、或是其他任何机制; 只要它遵守合约, 我们就不必在意

练习 15: 来自“按合约设计”(119 页)

下面, 让我们试一试一个 Java 的例子——与练习 14 中的例子有点类似。`insertNumber` 把整数插入有序列表中。前条件和后条件的标注方式与 `iContract` (参见[URL 17]) 一样。

```
private int data[];
/**
 * @post data[index-1] < data[index] &&
 *       data[index] == aValue
 */
public Node insertNumber (final int aValue)
{
  int index = findPlaceToInsert(aValue);
  ...
}
```

解答 15: 这是坏合约。下标子句中的计算 (`index-1`) 在边界条件上 (比如第一项) 无法工作

后条件假定了特定的实现: 我们想要合约比这更抽象

练习 16：来自“按合约设计”（119 页）

下面的代码段来自 Java 的栈类。这是好合约吗？

```
/**
 * @pre anitem != null      Require real data
 * @post pop() == anitem   / Verify that it's
 *                          / on the stack
 */
public void push(final String anItem)
```

解答 16：这是好合约，但却是坏实现。在这里，臭名昭著的“海森堡虫子”[URL 52]冒出了难看的头。程序员可能只是犯了一个简单的击键错误——把 top 敲成了 pop。尽管这是一个简单的、人为制造的例子，断言（或代码中任何意外的位置）中的副作用可能会非常难以诊断。

练习 17：来自“按合约设计”（119 页）

DBC 的经典例子（如练习 14-16 中的例子）给出的是某种 ADT（Abstract Data Type）的实现——栈或队列就是典型的例子。但并没有多少人真的会编写这种低级的类。

所以，这个练习的题目是，设计一个厨用搅拌机接口。它最终将是一个基于 Web、适用于 Internet、CORBA 化的搅拌机，但现在我们只需要一个接口来控制它。它有十挡速率设置（0 表示关机）。你不能在它空的时候进行操作，而且你只能一挡一挡地改变速率（也就是说，可以从 0 到 1，从 1 到 2，但不能从 0 到 2）。

下面是各个方法。增加适当的前条件、后条件和不变项。

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

解答 17：我们将给出用 Java 编写的函数型构（signature），并按 iContract 的规定标出前条件和后条件。

首先，类的不变项是：

```
/**
 * @invariant getSpeed() >= 0
 *           implies isFull()           // Don't run empty
 * @invariant getSpeed() <= 10 &&
 *           getSpeed() < 10           // Range check
 */
```

然后，前条件和后条件是：

```
/**
```



```

    * @pre Math.abs(getSpeed() - x) <= 1 // Only change by one
    * @pre x >= 0 && x < 10             // Range check
    * @post getSpeed() == x             // Honor requested speed
    */
    public void setSpeed(final int x)
    /**
     * @pre !isFull()                     // Don't fill it twice
     * @post isFull()                     // Ensure it was done
     */
    void fill()
    /**
     * @pre isFull()                       // Don't empty it twice
     * @post !isFull()                     // Ensure it was done
     */
    void empty()

```

练习 18: 来自“按合约设计”(119 页)

在 0, 5, 10, 15, ..., 100 序列中有多少个数?

解答 18: 在序列中有 21 个数。如果说是 20, 你就是在经验篱笆桩错误

练习 19: 来自“断言式编程”(125 页)

一次快速的真实性检查。下面这些“不可能”的事情中, 那些可能发生?

1. (标准阳历) 一个月少于 28 天
2. `stat(".", &sb) == -1` (也就是, 无法访问当前目录)
3. 在 C++ 里: `a = 2; b = 3; if (a + b != 5) exit(1);`
4. 内角和不同于 180° 的三角形
5. 没有 60 秒的一分钟
6. 在 Java 中: `(a + 1) <= a`

解答 19:

1. 1752 年 9 月只有 19 天。这是作为“格里高利改革”的一部分、为使日历同步而进行的。
2. 目录可能已被另外的进程移除、你可能没有权限读取它、&sb 可能无效——你知道意思了吗?
3. 我们偷偷地没有规定 `a` 和 `b` 的类型。操作符重载可能已把 `+`、`=` 或 `!=` 定义成具有意想不到的行为。而且, `a` 和 `b` 可能是同一变量的别名, 于是第二次赋值将覆盖第一次赋值所存储的值。
4. 在非欧几何中, 三角形的内角和不同于 180° 。想一想投影在球面上的三角形。

5. 闰分可能有 61 或 62 秒。
6. 溢出可能会使 `a+1` 的结果为负（在 C 和 C++ 里也可能发生这种情况）

练习 20：来自“断言式编程”（125 页）

为 Java 开发一个简单的断言检查类

解答 20：我们选择实现一个非常简单的类，它只有一个静态方法：`TEST`。如果传递的 `condition` 参数为假，它就打印一条消息和栈踪迹

```
package com.pragprog.util;

import java.lang.System;      // for exit()
import java.lang.Thread;      // for dumpStack()
public class Assert {
    /** Write a message, print a stack trace and exit if
     * our parameter is false.
     */
    public static void TEST(boolean condition) {
        if (!condition) {
            System.out.println("==== Assertion Failed ====");
            Thread.dumpStack();
            System.exit(1);
        }
    }
    // Testbed. If our argument is 'okay', try an assertion that
    // succeeds, if 'fail' try one that fails
    public static final void main(String args[]) {
        if (args[0].compareTo("okay") == 0) {
            TEST(1 == 1);
        }
        else if (args[0].compareTo("fail") == 0) {
            TEST(1 == 2);
        }
        else {
            throw new RuntimeException("Bad argument");
        }
    }
}
```

练习 21：来自“何时使用异常”（128 页）

在设计一个新的容器类时，你确定可能有以下错误情况：

1. `add` 例程中的新元素没有内存可用

2. 在 fetch 例程中找不到所请求的数据项
3. 传给 add 例程的是 null 指针

应怎样处理每种情况？应该生成错误、引发异常、还是忽略该情况？

解答 21：用光内存是一种异常情况，所以我们觉得（1）应该引发异常

没能找到数据项很可能是正常情况。调用我们的集合类（collection class）的应用可能已写好了代码，在增加潜在的重复项之前检查数据项是否已经存在。我们觉得（2）应该只是返回错误指示

（3）更成问题——如果值 null 对应用有意义，那么把它加入容器中就是正当的。但是，如果存储 null 值没有意义，可能就应该抛出异常

练习 22：来自“怎样配平资源”（136 页）

有些 C 和 C++ 开发者故意在解除了某个指针引用的内存的分配之后，把该指针设为 NULL。这为什么是个好主意？

解答 22：在大多数 C 和 C++ 实现中，没有检查指针实际上是否指向有效内存的途径。一种常见的错误是：解除了某个内存块的分配，随后又在程序中引用该内存。此时，该指针所指向的内存很可能已重新分配用于其他目的。通过把指针设置为 NULL，程序员希望能预防这些有害的引用——在大多数情况下，解除 NULL 指针的引用将生成运行时错误。

练习 23：来自“怎样配平资源”（136 页）

有些 Java 开发者故意在使用完某个对象之后，把该对象变量设为 NULL，这为什么是个好主意？

解答 23：通过把引用设置为 NULL，你使指向被引用对象的指针数目减一。一旦此计数达到零，对象就会符合垃圾收集的条件。把引用设置为 NULL 对长期运行的程序可能会有重要意义，在这样的程序里，程序员需要确保内存使用不会随时间而增长。

练习 24：来自“解耦与得墨忒耳法则”（143 页）

我们在上一页的方框中讨论了物理解耦。下面的 C++ 头文件中，哪一个与系统的其余部分更紧密地耦合在一起？

person1.h: <pre>#include "date.h" class Person1 { private: Date myBirthdate; public: Person1(Date &birthdate); // ...</pre>	person2.h <pre>class Date; class Person2 { private: Date *myBirthdate; public: Person2(Date &birthDate); // ...</pre>
--	--

解答 24：头文件的用途是定义相应实现与其他部分之间的接口。头文件自身无需了解 Date 类的内部情况——它只需告诉编译器，构造器用 Date 对象做参数。所以，第二段代码能很好地工作，除非头文件在 inline 函数中使用了 Date。

第一段代码有什么问题？对于小项目，除了你不必要地让使用 Person1 类的所有程序都包含 Date 的头文件之外，没有什么问题。一旦这种做法在项目中变得很常见，你很快就会发现，包含一个头文件最终将会包含系统其他的大部分头文件——这会严重地拖长编译时间。

练习 25：来自“解耦与得墨忒耳法则”（143 页）

对于下面的以及练习 26 和 27 中的例子，根据得墨忒耳法则，确定所示方法调用是否允许。第一个例子是用 Java 编写的。

```
public void showBalance(BankAccount acct) {
    Money amt = acct.getBalance();
    printToScreen(amt.printFormat());
}
```

解答 25：变量 acct 是作为参数传入的，所以允许调用 getBalance。但是，调用 amt.printFormat() 却非如此。我们并不“拥有” amt，它也不是传给我们的。通过下面的代码，我们可以消除 showBalance 与 Money 的耦合。

```
void showBalance(BankAccount b) {
    b.printBalance();
}
```

练习 26: 来自“解耦与得墨忒耳法则”(143 页)

这个例子也是用 Java 编写的。

```
public class Colada {
    private Blender myBlender;
    private Vector myStuff;
    public Colada() {
        myBlender = new Blender();
        myStuff = new Vector();
    }
    private void doSomething() {
        myBlender.addIngredients(myStuff.elements());
    }
}
```

解答 26: 因为 Colada 创建并拥有 myBlender 和 myStuff, 所以可以调用 addIngredients 和 elements。

练习 27: 来自“解耦与得墨忒耳法则”(143 页)

这个例子是用 C++ 编写的。

```
void processTransaction(BankAccount acct, int) {
    Person *who;
    Money amt;

    amt.setValue(123.45);
    acct.setBalance(amt);
    who = acct.getOwner();
    markWorkflow(who->name(), SET_BALANCE);
}
```

解答 27: 在这个例子中, processTransaction 拥有 amt——它是在栈上创建的, acct 是传入的, 所以既可以调用 setValue, 也可以调用 setBalance。但 processTransaction 并不拥有 who, 所以调用 who->name() 违反了法则。得墨忒耳法则建议把这行代码改写成:

```
markWorkflow(acct.name(), SET_BALANCE);
```

processTransaction 没有必要了解是 BankAccount 中的哪个子对象持有账户名——这一结构知识不应通过 BankAccount 的合约显露出来。相反, 我们请求 BankAccount 提供账户名。它知道自己把账户名放在何处(也许是在 Person 中、在 Business 中、或是在多态的 Customer 对象中)

练习 28：来自“元程序设计”（149 页）

下面的哪些事物最好表示为程序中的代码，哪些最好表示为外部的元数据？

1. 通信端口指派
2. 编辑器对各种语言的语法突显的支持
3. 编辑器对不同的图形设备的支持
4. 解析器或扫描器的状态机
5. 用于单元测试的样本值和结果

解答 28：在这里没有确定的答案——问题主要是为了让你思考而提出的。但是，下面是我们的看法：

1. **通信端口指派** 显然，这一信息应该存储为元数据。但详细到什么程度？有些 Windows 程序只让你选择波特率和端口（比如 COM1 到 COM4）但也许你还需要指定字长、奇偶位、停止位、以及双工设置。在可行的情况下，设法存储最详细的细节。
2. **编辑器对各种语言的语法突显的支持** 这应该实现为元数据。你不会希望自己必须去改动代码，只是因为 Java 的最新版本引入了一个新关键字。
3. **编辑器对不同的图形设备的支持** 这可能难以严格地实现为元数据。你不会想让多种图形驱动程序压在你的应用上，却只是为了在运行时选择某一种。但是，你可以用元数据来指定驱动程序的名称，并动态加载代码。这是用人能够阅读的格式存放元数据的另一个好论据；如果你用程序设置了一种功能失常的视频驱动程序，你可能无法用程序把它设回来。
4. **解析器或扫描器的状态机**。这取决于你在解析或扫描什么。如果你在解析某种由标准组织严格定义的数据，并且没有国会的法令不大可能改变，那么对其进行硬编码就没有问题。但如果你面临的是更为易变的情形，那么在外部分定义状态表可能会有好处。
5. **用于单元测试的样本值和结果** 大多数应用在测试装备内部定义这些值，但把测试数据——还有对可接受的结果的定义——移到代码自身之外，你可以获得更好的灵活性。

练习 29: 来自“它只是视图”(164 页)

假定你有一个航空订座系统，其中包含有这样的航班概念：

```
public interface Flight {
    // Return false if flight full.
    public boolean addPassenger(Passenger p);
    public void addWaitList(Passenger p);
    public int getFlightCapacity();
    public int getNumPassengers();
}
```

如果你把乘客加入等候名单，他们将在有空座时被自动加入该航班

有一个大型的报表作业负责查看超额订座或是满员的航班，以提议何时增开航班。它工作得很好，但运行时间却需要数小时。

我们想在处理等候名单时多一点灵活性，我们必须对那个大报表做点什么——它的运行时间太长了。使用这一节介绍的思想重新设计这个接口。

解答 29: 我们将给 `Flight` 增加另外一些方法，用于维护两个侦听者 (listener) 列表：一个用于等候名单通知，另一个用于航班满员通知。

```
public interface Passenger {
    public void waitListAvailable();
}

public interface Flight {
    ...
    public void addWaitListListener(Passenger p);
    public void removeWaitListListener(Passenger p);
    public void addFullListener(FullListener b);
    public void removeFullListener(FullListener b);
    ...
}

public interface BigReport extends FullListener {
    public void FlightFullAlert(Flight f);
}
```

如果我们在试图增加 `Passenger` 时因航班满员而失败，我们可以(可选地)把 `Passenger` 放入等候名单。出现空座时，`waitListAvailable` 会被调用。该方法可以随即选择自动增加 `Passenger`，或是让服务代表打电话给客户，询问他们是否仍然有意订座，或是否有其他需要。我们现在可以灵活地按照每个客户的需要进行不同的处理。

接下来，我们想要避免让 `BigReport` 通过检查大量记录搜寻满员的航班。通过让 `BigReport` 作为侦听者向 `Flight` 注册，各个 `Flight` 可以在满员时(或接近满

员时，如果我们想这么做）发出报告。现在，用户可以即时地得到 BigReport 的正在发生的、最新的报告，而不需要像以前那样等待它运行几个小时。

练习 30：来自“黑板”（170 页）

对于下面的各个应用，黑板系统是否适用？为什么？

1. **图像处理**。你想要让一些并行进程抓取图像块，进行处理，并把完成后的图像块放回去。
2. **机构日程安排**。你的员工分散在全球，在不同的时区，并且说不同的语言，你要安排一次会议。
3. **网络监控工具**。系统搜集性能统计信息，并收集问题报告。你想要实现一些代理，使用这些信息查找系统中的问题。

解答 30：

1. **图像处理**。对于简单的并行进程间的工作负载调度、共享的工作队列也许就很足够了。如果涉及反馈——也就是说，如果某块图像的处理结果对其它块有影响，就像是机器视觉应用或复杂的 3D 图像扭曲变换中的情况——你可以考虑黑板系统。
2. **机构日程安排**。黑板系统也许很适用。你可以在黑板上张贴会议安排及有效性。你有一些独立运作的机构，对决策的反馈很重要，并且参与者可以来来去去。

你可以考虑根据搜索人划分这种黑板系统：初级职员可能只关心临近的办公室，人力资源部可能只想要世界各地说英语的办公室，而 CEO 可能想要全部资料。

在数据格式上也很灵活：我们有忽略我们不理解的格式或语言的自由。我们只需理解那些要一起开会的办公室的不同格式，而且我们不需要让所有的参与者都参与所有可能格式的完整的“传递闭包”（transitive closure）。这使耦合减到了必需的程度，并且不会人为地束缚我们。

3. **网络监控工具**。这与 168 页描述的抵押/贷款应用程序非常类似。你收到用户发来的问题报告和自动报告的统计信息，它们都公布在了黑板上。人或软件代理可以分析黑板，诊断网络故障：线路上的两个错误也许只是宇宙射线造成的，但 20 000

个错误就是硬件问题。就像侦探侦破神秘的谋杀案一样，你可以让多个实体进行分析、为解决网络问题出主意。

练习 31：来自“靠巧合编程”（176 页）

你能否识别出下面的 C 代码段中的一些巧合？假定这段代码深埋在某个库例程中。

```
fprintf(stderr, "Error, continue?");
gets(buf);
```

解答 31：这段代码有若干潜在的问题。首先，它假定使用的是 tty 环境。如果假定成立，这没有问题，但如果是从 GUI 环境调用这段代码，stderr 和 stdin 都没有打开呢？

其次，gets 也成问题，它会把接收到的所有字符全都写入传入的缓冲区。恶意用户一直在利用这一缺陷，在许多不同的系统中制造缓冲区溢出安全漏洞。永远不要使用 gets()。

第三，代码假定用户理解英语。

最后，心智正常的人决不会把这样的用户交互代码埋在库例程中。

练习 32：来自“靠巧合编程”（176 页）

这段 C 代码有时在有些机器上能工作，但有时又不能。有什么问题？

```
/* Truncate string to its last maxlen chars *.
void string_tail(char *string, int maxlen) {
    int len = strlen(string);
    if (len > maxlen) {
        strcpy(string, string + (len - maxlen));
    }
}
```

解答 32：对于重叠的字符串，POSIX 的 strcpy 不保证能正常工作。在有些架构上它也许碰巧能工作，但那只是巧合。

练习 33：来自“靠巧合编程”（176 页）

这段代码来自某个通用的 Java 跟踪套件。该函数把一个字符串写到日志文件中。它通过了单元测试，但当一个 Web 开发者使用它时却失败了。它依靠了什么巧合？

```
public static void debug(String s) throws IOException {  
    FileWriter fw = new FileWriter("debug.log", true);  
    fw.write(s);  
    fw.flush();  
    fw.close();  
}
```

解答 33：在有禁止写本地磁盘的安全约束的 applet 语境中，这段代码无法工作。再一次，如果你要选择是否在 GUI 语境中运行，你可以动态地查看当前的环境是什么样的。在这个例子中，如果不能访问本地磁盘，你可以把日志文件放在别的地方。

练习 34：来自“算法速率”（183 页）

我们编写了一组简单的排序例程，可从我们的网站（www.pragmaticprogrammer.com）下载。在你可以使用的各种机器上运行它们。你的数字是否遵循预期的曲线？关于你的机器的相对速度，你可以推断出什么？各种编译器优化设置的效果是什么？基数排序真的是线性的吗？

解答 34：显然，我们不能就这一问题给出绝对的答案。但是，我们可以给你一些提示。

如果你发现你的结果不是平滑的曲线，你可以查看是否有其他活动在使用你的处理器。在多用户系统上，你大概无法得到好的数字，而且即使你是惟一的用户，你可能也会发现后台进程在周期性地拿走你的程序的处理周期。你还可以检查内存：如果应用开始使用交换空间，性能就会急剧下降。

用不同的编译器和不同的优化设置进行试验会很有趣。我们发现，启用积极的优化，有可能获得令人吃惊的速率提升。我们还发现，在更广泛的 RISC 架构上，制造商的编译器常常要胜过可移植性更好的 GCC。大概，制造商私底下知道在这些机器上生成高效代码的秘密。

练习 35：来自“算法速率”（183 页）

下面的例程打印出二叉树的内容。假定树是平衡的，例程在打印一棵有 1 000 000 个元素的树时大约要使用多少栈空间？（假定子例程调用不会带来显著的栈开销）

```
void printTree(const Node *node) {
    char buffer[1000];
    if (node) {
        printTree(node->left);
        getNodeAsString(node, buffer);
        puts(buffer);
        printTree(node->right);
    }
}
```

解答 35：printTree 例程把大约 1 000 字节的栈空间用于 buffer 变量。它递归地调用自己，下行遍历二叉树，每个嵌套的调用又使用 1 000 字节的栈空间。当它到达叶节点时也会调用自己，但在发现传入的是 NULL 指针时会立即退出。因此，如果树的深度是 D ，最大的栈需求大约就是 $1000 \times (D + 1)$ 。

平衡的二叉树的每一层的元素都比上一层多一倍。深度为 D 的树持有 $1 + 2 + 4 + 8 + \dots + 2^{(D-1)}$ （即 $2^D - 1$ ）个元素。因此我们的有百万元素的树将需要 $\lceil \lg(1,000,001) \rceil$ 层，即 20 层。

因此我们预计，我们的例程将使用大约 21 000 字节栈空间。

练习 36：来自“算法速率”（183 页）

（除了减小缓冲区）你能否想出任何减少练习 35 中的例程的栈需求的方法？

解答 36：我们想到了几种优化方法。首先，当 printTree 例程在叶节点上调用自身时，会立即退出，因为没有孩子。这样的调用使最大栈空间增加了约 1 000 字节。我们还可以消除尾递归（第二次递归调用），尽管这对最坏情况下的栈使用没有影响。

```
while (node) {
    if (node->left) printTree(node->left);
    getNodeAsString(node, buffer);
    puts(buffer);
    node = node->right;
}
```

但是，最大的收益来自只分配一个缓冲区，由对 `printTree` 的所有调用共用。把这个缓冲区作为参数传给递归的调用，不管递归有多深，都只要分配 1 000 字节。

```
void printTreePrivate(const Node *node, char *buffer) {
    if (node) {
        printTreePrivate(node->left, buffer);
        getNodeAsString(node, buffer);
        puts(buffer);

        printTreePrivate(node->right, buffer);
    }
}

void newPrintTree(const Node *node) {
    char buffer[1000];
    printTreePrivate(node, buffer);
}
```

练习 37：来自“算法速率”（183 页）

在 180 页，我们宣称二分法是 $O(\ln(n))$ 。你能证明吗？

解答 37：有几种证明方式。一种是回到问题的开头。如果数组只有一个元素，我们不用迭代循环。每多一次迭代，我们可以查找的数组的大小都会加倍。因此，数组大小的一般公式是 $n = 2^m$ ，其中 m 是迭代次数。如果你对公式两边取底为 2 的对数，就得到 $\lg(n) = \lg(2^m)$ ，按照对数的定义就变成了 $\lg(n) = m$ 。

练习 38：来自“重构”（188 页）

下面的代码显然在几年里进行了数次更新，但所做改动并未改善其结构。重构它

```
if (state == TEXAS) {
    rate = TX_RATE;
    amt = base * TX_RATE;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
else if ((state == OHIO) || (state == MAINE;
    rate = (state == OHIO) ? OH_RATE : ME_RATE;
    amt = base * rate;
    calc = 2*basis(amt) + extra(amt)*1.05;
    if (state == OHIO)
        points = 2;
}
else {
    rate = 1;
    amt = base;
    calc = 2*basis(amt) + extra(amt)*1.05;
}
```

解答 38: 在这里我们想提出相当温和的重构：确保每个测试只进行一次，并且使所有的计算成为共用的。如果表达式 `2 * basis(...) * 1.05` 也出现在程序中别的地方，我们可能应该使它成为函数。在这里我们没有这样做。

我们增加了一个 `rate_lookup` 数组，并进行初始化，使除 Texas、Ohio 和 Maine 之外的项的值都为 1。这种方法使得我们很容易在未来增加其他州的值。取决于所期望的使用模式，我们可以使 `points` 也成为能够进行数组查找的字段。

```
rate = rate_lookup[state];
amt = base * rate;
calc = 2*basis(amt) + extra(amt)*1.05;
if (state == OHIO)
    points = 2;
```

练习 39: 来自“重构”（188 页）

下面的 Java 类需要支持更多的形状。重构这个类，为增加做准备。

```
public class Shape {

    public static final int SQUARE    = 1;
    public static final int CIRCLE    = 2;
    public static final int RIGHT_TRIANGLE = 3;

    private int shapeType;
    private double size;
    public Shape(int shapeType, double size) {
        this.shapeType = shapeType;
        this.size = size;
    }
    // ... other methods ...
    public double area(){
        switch (shapeType) {
            case SQUARE:    return size*size;
            case CIRCLE:    return Math.PI*size*size/4.0;
            case RIGHT_TRIANGLE: return size*size/2.0;
        }
        return 0;
    }
}
```

解答 39: 当你看到有人使用枚举类型（或其 Java 等价物）区分某种类型的各种变体时，你常常可以通过定义子类来改善代码：

```
public class Shape {
    private double size;
    public Shape(double size) {
        this.size = size;
    }
    public double getSize() { return size; }
}
public class Square extends Shape {
    public Square(double size) {
        super(size);
    }
    public double area() {
        double size = getSize();
        return size*size;
    }
}
public class Circle extends Shape {
    public Circle(double size) {
        super(size);
    }
    public double area() {
        double size = getSize();
        return Math.PI*size*size/4.0;
    }
}
// etc...
```

练习 40: 来自“重构”（188 页）

这段 Java 代码是将贯穿你的项目使用的某个框架的组成部分 进行重构，让其更一般化，并且将来更易于扩展。

```
public class Window {
    public Window(int width, int height) { ... }
    public void setSize(int width, int height) { ... }
    public boolean overlaps(Window w) { ... }
    public int getArea() { ... }
}
```

解答 40: 这个类很有趣。初看上去，窗口有宽度和高度似乎是合理的。但是，考虑一下未来——让我们想象一下，我们想要支持任意形状的窗口（如果 Window 类只知道矩形及其属性，这将很困难）。

我们提议把窗口的形状从 Window 类自身中抽取出来。

```
public abstract class Shape {
    // ...
    public abstract boolean overlaps(Shape s);
    public abstract int getArea();
}
public class Window {

    private Shape shape;

    public Window(Shape shape) {
        this.shape = shape;
        ...
    }
    public void setShape(Shape shape) {
        this.shape = shape;
        ...
    }
    public boolean overlaps(Window w) {
        return shape.overlaps(w.shape);
    }
    public int getArea() {
        return shape.getArea();
    }
}
```

注意在此方法中，我们使用了委托（delegation），而不是子类定义：窗口不是一种“形状”——窗口“有”形状。它使用形状来完成它的工作。在重构时，你常常会发现委托很有用。

我们还可以这样扩展这个例子：引入一个 Java 接口，规定一个类为了支持形状功能、必须支持的方法。这是一个好主意。这意味着，当你扩展形状的概念时，编译器将会就你影响到的类向你发出警告。我们建议你在委托其它类的所有函数时这样使用接口。

练习 41：来自“易于测试的代码”（197 页）

为练习 17 的解答中描述的搅拌机接口（289 页）设计一个测试用具。编写一个 shell 脚本，对搅拌机进行回归测试。你需要测试基本功能、错误和边界条件、以及合约规定的任何义务。对速度改变有何限制？这些限制得到了遵守吗？

解答 41：首先，我们将增加 main，充当单元测试驱动程序。它将接受一种非常小的

简单语言作为参数：“E” 清空搅拌机、“F” 装填、0-9 设置速度，等等

```
public static void main(String args[]) {
    // Create the blender to test
    dbc_ex blender = new dbc_ex();
    // And test it according to the string on standard input
    try {
        int a;
        char c;
        while ((a = System.in.read()) != -1) {

            c = (char)a;

            if (Character.isWhitespace(c)) {
                continue;
            }
            if (Character.isDigit(c)) {
                blender.setSpeed(Character.digit(c, 10));
            }
            else {
                switch (c) {
                    case 'F': blender.fill();
                        break;
                    case 'E': blender.empty();
                        break;
                    case 's': System.out.println("SPEED: " +
                        blender.getSpeed());
                        break;
                    case 'f': System.out.println("FULL " +
                        blender.isFull());
                        break;
                    default: throw new RuntimeException(
                        "Unknown Test directive");
                }
            }
        }
    }
    catch (java.io.IOException e) {
        System.err.println("Test jig failed: " + e.getMessage());
    }
    System.err.println("Completed blending\n");
    System.exit(0);
}
```

下面是驱动测试的 shell 脚本。

```
#!/bin/sh
CMD="java dbc.dbc_ex"
failcount=0
expect_okay() {
    if echo "$*" | $CMD #>/dev/null 2>&1
```



```

    then
    :
    else
        echo "FAILED! $*"
        failcount='expr $failcount + 1'
    fi
}
expect_fail() {
    if echo "$*" | $CMD >/dev/null 2>&1
    then
        echo "FAILED! (Should have failed): $*"
        failcount='expr $failcount + 1'
    fi
}
report() {
    if [ $failcount -gt 0 ]
    then
        echo -e "\n\n*** FAILED $failcount TESTS\n"
        exit 1 # In case we are part of something larger
    else
        exit 0 # In case we are part of something larger
    fi
}
#
# Start the tests
#
expect_okay F123456789876543210E # Should run thru
expect_fail F5 # Fails, speed too high
expect_fail[] # Fails, empty
expect_fail F10E1 # Fails, empty
expect_fail F1238 # Fails, skips
expect_okay FE # Never turn on
expect_fail F1E # Emptying while running
expect_okay F10E Should be ok
report # Report results

```

这些测试查看是否检测到非法的速度改变、是否试图在搅拌机正运转时将其清空，等等。我们把它放在 `makefile` 中，这样我们简单地敲入下面的命令就可以编译和运行回归测试：

```

% make
% make test

```

注意，我们让测试退出时返回 0 或 1，这样我们还可以将其用作更大测试的一部分。

在需求中没有说要通过脚本、甚或语言来驱动这个组件。最终用户永远也不会看

到它。但我们拥有了一个强大的工具，可以用来快速而详尽地测试我们的代码。

练习 42：来自“需求之坑”（211 页）

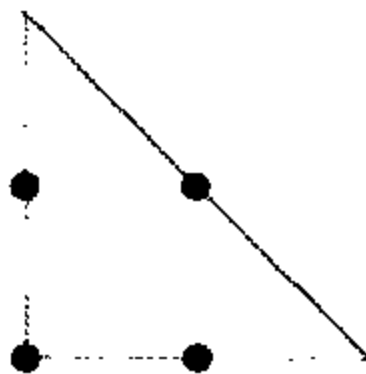
下面各项中，哪些可能是真正的需求？（如果可能）重新陈述那些并非真正的需求的项。

1. 响应时间必须小于 500ms
2. 对话框背景为灰色。
3. 应用将被组织成许多前端进程和一个后端服务器。
4. 如果用户在数字字段中输入非数字字符，系统将发出蜂鸣声，并且不接受它们。
5. 应用代码和数据必须不超出 256KB。

解答 42：

1. 这个陈述听起来像是真实的需求：可能存在由应用的环境施加给应用的约束。
2. 即使这可能是公司标准，也不是需求。这样陈述可能更好：“对话框背景必须可由最终用户配置。在发运时，其颜色将是灰色。”下面的更宽泛的陈述更好：“应用的所有可视元素（颜色、字体、语言）必须可由最终用户配置。”
3. 这个陈述不是需求，是架构。在面对这样的陈述时，你必须深入了解用户的所思所想。
4. 在其之下的需求可能接近于“系统将防止用户在字段中输入无效的数据项，并在发生这样的情况时警告用户。”
5. 这个陈述很可能是硬性需求。

213 页的“四柱谜题”的一种解法。



A

- Accessor function, 31
- ACM, *see* Association for Computing Machinery
- Active code generator, 104
- Activity diagram, 150
- Advanced C++ Programming Styles and Idioms*, 265
- Advanced Programming in the Unix Environment*, 264
- Aegis transaction-based configuration management, 246, 271
- Agent, 76, 117, 297
- Algorithm
 - binary chop, 180
 - choosing, 182
 - combinatoric, 180
 - divide-and-conquer, 180
 - estimating, 177, 178
 - linear, 177
 - $O()$ notation, 178, 181
 - quicksort, 180
 - runtime, 181
 - sublinear, 177
- Allocations, nesting, 131
- Analysis Patterns*, 264
- Anonymity, 258
- AOP, *see* Aspect-Oriented Programming
- Architecture
 - deployment, 156
 - flexibility, 46
 - prototyping, 55
 - temporal decoupling, 152
- Art of Computer Programming*, 183
- Artificial Intelligence, marauding, 26
- Aspect-Oriented Programming (AOP), 39, 273
- Assertion, 113, 122, 175
 - side effects, 124

- turning off, 123
- Association for Computing Machinery (ACM), 262
 - Communications of the ACM*, 263
 - SIGPLAN*, 263
- Assumptions, testing, 175
- "at" command, 231
- Audience, 21
 - needs, 19
- auto_ptr, 134
- Automation, 230
 - approval procedures, 235
 - build, 88, 233
 - compiling, 232
 - cron, 231
 - documentation, 251
 - scripts, 234
 - team, 229
 - testing, 29, 238
 - Web site generation, 235
- awk, 99

B

- Backus-Naur Form (BNF), 59n
- Base class, 112
- bash shell, 80, 82n
- Bean, *see* Enterprise Java Beans (EJB)
- Beck, Kent, 194, 258
- Beowulf project, 268
- "Big O" notation, 177
- "Big picture", 8
- Binary chop, 97, 180
- Binary format, 73
 - problems parsing, 75
- bison, 59, 269
- BIST, *see* Built-In Self Test
- Blackboard system, 165
 - partitioning, 168
 - workflow, 169

Blender example
 contract for, 119, 289
 regression test jig, 305
 workflow, 151
 BNF, *see* Backus-Naur Form (BNF)
 Boiled frog, 8, 175, 225
 Boundary condition, 173, 243
 Brain, Marshall, 265
 Branding, 226
 Brant, John, 268
 "Broken Window Theory", 5
 vs. stone soup, 9
 Brooks, Fred, 264
 Browser, class, 187
 Browser, refactoring, 187, 268
 Bug, 90
 failed contract *as*, 111
 see also Debugging; Error
 Build
 automation, 88, 233
 dependencies, 233
 final, 234
 nightly, 231
 refactoring, 187
 Built-In Self Test (BIST), 189
 Business logic, 146
 Business policy, 203

C

C language
 assertions, 122
 DBC, 114
 duplication, 29
 error handling, 121
 error messages, 115
 macros, 121
 Object Pascal interface, 101
 C++ language, 46
 assertions, 122
 auto_ptr, 134
 books, 265
 DBC, 114
 decoupling, 142
 DOC++, 251, 269
 duplication, 29
 error messages, 115
 exceptions, 132
 unit tests, 193
 Caching, 31

Call, routine, 115, 173
 Cascading Style Sheets (CSS), 253
 Cat
 blaming, 3
 herding, 224
 Schrödinger's, 47
 Catalyzing change, 8
 Cathedrals, xx
 Cetus links, 265
 Change, catalyzing, 8
 Christiansen, Tom, 81
 Class
 assertions, 113
 base, 112
 coupling, 139, 142
 coupling ratios, 242
 encapsulating resource, 132
 invariant, 110, 113
 number of states, 245
 resource allocation, 132
 subclass, 112
 wrapper, 132, 133, 135, 141
 Class browser, 187
 ClearCase, 271
 Cockburn, Alistair, *xxii*, 205, 264, 272
 Code generator, 28, 102
 active, 104
 makefiles, 232
 parsers, 105
 passive, 103
 Code profiler, 182
 Code reviews, 33, 236
 Coding
 algorithm speed, 177
 comments, 29, 249
 coupled, 130
 coverage analysis, 245
 database schema, 104
 defensive, 107
 and documentation, 29, 248
 estimating, 68
 exceptions, 125
 implementation, 173
 iterative, 69
 "lazy", 111
 metrics, 242
 modules, 138
 multiple representations, 28
 orthogonality, 34, 36, 40

- ownership, 258
 - prototypes, 55
 - server code, 196
 - "shy", 40, 138
 - specifications, 219
 - tracer bullets, 49–51
 - unit testing, 190, 192
 - see also Coupled code; Decoupled code; Metadata; Source code control system (SCCS)
 - Cohesion, 35
 - COM, see Component Object Model
 - Combinatorial explosion, 140, 167
 - Combinatoric algorithm, 180
 - Command shell, 77
 - bash, 80
 - Cygwin, 80
 - vs. GUI, 78
 - UWIN, 81
 - Windows, 80
 - Comment, 29, 249
 - avoiding duplication, 29
 - DBC, 113
 - parameters, 250
 - types of, 249
 - unnecessary, 250
 - see also Documentation
 - Common Object Request Broker (CORBA), 29, 39, 46
 - Event Service, 160
 - Communicating, 18
 - audience, 19, 21
 - duplication, 32
 - e-mail, 22
 - and formal methods, 221
 - presentation, 20
 - style, 20
 - teams, 225
 - users, 256
 - writing, 18
 - Communications of the ACM*, 263
 - Comp.object FAQ, 272
 - Compiling, 232
 - compilers, 267
 - DBC, 113
 - warnings and debugging, 92
 - Component Object Model (COM), 55
 - Component-based systems, see Modular system
 - Concurrency, 150
 - design, 154
 - interfaces, 155
 - and Programming by Coincidence, 154
 - requirements analysis of, 150
 - workflow, 150
 - Concurrent Version System (CVS), 271
 - Configuration
 - cooperative, 148
 - dynamic, 144
 - metadata, 147
 - Configuration management, 86, 271
 - Constantine, Larry L., 35
 - Constraint management, 213
 - Constructor, 132
 - initialization, 155
 - Contact, authors' e-mail, xxiii
 - Context, use instead of globals, 40
 - Contract, 109, 174
 - see also Design by contract (DBC)
 - Controller (MVC), 162
 - Coplien, Jim, 265
 - CORBA, see Common Object Request Broker
 - Coupled code, 130
 - coupling ratios, 242
 - minimizing, 138, 158
 - performance, 142
 - temporal coupling, 150
 - see also Decoupled code
 - Coverage analysis, 245
 - Cox, Brad J., 189n
 - Crash, 120
 - Critical thinking, 16
 - cron, 231
 - CSS, see Cascading Style Sheets
 - CVS, see Concurrent Version System
 - Cygwin, 80, 270
- ## D
-
- Data
 - blackboard system, 169
 - caching, 31
 - dictionary, 144
 - dynamic data structures, 135
 - global, 40
 - language, 60
 - normalizing, 30

- readable vs. understandable, 75
- test, 100, 243
- views, 160
- visualizing, 93
 - see also Metadata
- Data Display Debugger (DDD), 93, 268
- Database
 - active code generator, 104
 - schema, 105f, 141, 144
 - schema maintenance, 100
- DBC, see Design by contract
- DDD, see Data Display Debugger
- Deadline, 6, 246
- Deadlock, 131
- Debugging, 90
 - assertions, 123
 - binary search, 97
 - bug location, 96
 - bug reproduction, 93
 - checklist, 98
 - compiler warnings and, 92
 - corrupt variables, 95
 - "Heisenbug", 124
 - rubber ducking, 95
 - and source code branching, 87
 - surprise bug, 97
 - and testing, 92, 195
 - time bomb, 192
 - tracing, 94
 - view, 164
 - visualizing data, 93
- Decision making, 46
- Decoupled code, 38, 40
 - architecture, 152
 - blackboard system, 166
 - Law of Demeter, 140
 - metadata, 145
 - minimizing coupling, 138
 - modular testing, 244
 - physical decoupling, 142
 - temporal coupling, 150
 - workflow, 150
 - see also Coupled code
- Defensive coding, 107
- Delegation, 304
- Delphi, 55
 - see also Object Pascal
- Demeter project, 274
- Demeter, Law of, 140
- Dependency, reducing, see Modular system; Orthogonality
- Deployment, 156
- Deployment descriptor, 148
- Design
 - accessor functions, 31
 - concurrency, 154
 - context, 174
 - deployment, 156
 - design/methodology testing, 242
 - metadata, 145
 - orthogonality, 34, 37
 - physical, 142
 - refactoring, 186
 - using services, 154
- Design by contract (DBC), 109, 155
 - and agents, 117
 - assertions, 113
 - class invariant, 110
 - as comments, 113
 - dynamic contracts, 117
 - lContract, 268
 - language support, 114
 - list insertion example, 110
 - pre- and postcondition, 110, 113, 114
 - predicates, 110
 - unit testing, 190
- Design Patterns, 264
 - observer, 158
 - singleton, 41
 - strategy, 41
- Destructor, 132
- Detectives, 165
- Development tree, 87
- Development, iterative, 69
- Divide-and-conquer algorithm, 180
- DOC++ documentation generator, 251, 269
- DocBook, 254
- Documentation
 - automatic updating, 251
 - and code, 29, 248
 - comments, 29, 113, 249, 251
 - executable, 251
 - formats, 253
 - HTML, 101
 - hypertext, 210
 - internal/external, 248

- invariant, 117
- mark-up languages, 254
- orthogonality, 42
- outline, 18
- requirements, 204
- technical writers, 252
- word processors, 252, 254
- writing specifications, 218
 - see also Comment; Web documentation
- Dodo, 148
- Domain, problem, 58, 66
- Don't repeat yourself, see DRY principle
- Downloading source code, see Example code
- Dr. Dobbs Journal*, 263
- DRY principle, 27, 29, 42
 - see also Duplication
- Duck, rubber, see Rubber duck
- Dumpty, Humpty, xxii, 165
- Duplication, 26
 - code generators avoid, 28
 - and code reviews, 33
 - design errors, 30
 - documentation and code, 29
 - DRY principle, 27, 29
 - interdeveloper, 32
 - in languages, 29
 - multiple representations, 28
 - teams, 226
 - under time pressure, 32
 - types of, 27
- Dynamic configuration, 144
- Dynamic data structure, 135
- Dynamics of Software Development*, 264

E

- E-mail, 22
 - address for feedback, xxiii
- Editor, 82
 - auto-indenting, 85
 - cursor movement, 84
 - features, 83
 - generating code, 103
 - how many to learn, 82
 - template, 84
 - types of, 266
 - Windows notepad, 84
- Effective C++*, 265
- Eiffel, 109, 114, 267
- EJB, see Enterprise Java Beans
- elvis editor, 267
- Emacs editor, 84, 266
 - Viper vi emulator, 267
- Embedded mini-language, 62, 145
- Embellishment, 11
- Encapsulation, object, 127, 158
- Eno, Brian, 205
- Enterprise Java Beans (EJB), 39, 147
- Entropy, 4
- Error
 - DBC messages, 115
 - design, 30
 - domain-specific, 59
 - early crash, 120
 - log messages, 196
 - orthogonality, 41
 - testing, 240, 247
 - see also Exception
- Error handler, 127
- Estimating, 64
 - accuracy, 64
 - algorithms, 177, 178
 - iterative, 69
 - models, 66
 - problem domain, 66
 - project schedules, 68
 - records, 68
 - testing, 182
- Eton College, xxi
- Event, 157
- Event channel, 160
- Example code
 - add logging, 40
 - airline reservations, 164, 296
 - assert macro, 122
 - auto_ptr example, 134
 - bad resource balancing, 129, 130
 - downloading, xxiii
 - exception error handling, 125
 - good resource balancing, 131
 - JavaDoc example, 250
 - method chaining, 139
 - normalized class, 31
 - open password file, 126
 - open user file, 127

- resources and exceptions, 132, 133
- side effect, 124
- spaghetti error handling, 125
- square root, 190
- string parsing with
 - StringTokenizer, 156
- string parsing with strtok, 155
- unnormalized class, 30

Example code by name

- AOP, 40
- Misc.java, 156
- assert, 122
- bad_balance.c, 129, 130
- balance.cc, 134
- balance.c, 131-133
- class Line, 30, 31
- exception, 125
- findPeak, 250
- interface Flight, 164, 296
- misc.c, 155
- openpasswd.java, 126
- openuserfile.java, 127
- plotDate, 139
- side_effect, 124
- spaghetti, 125
- sqrt, 190

Exception, 121

- effects of, 127
- and error handlers, 127
- missing files, 126
- resource balancing, 132
- when to use, 125

Excuses, 3

Executable document, 251

expect, 269

Expert, see Guru

Expiring asset, 12

eXtensible Style Language (XSL), 253

Extinction, 148

eXtreme Programming, 238n, 258, 272

F

Feature creep, 10

Feedback, e-mail address, xxiii

File

- exception, 126
- header, 29
- implementation, 29

- log, 196
- makefile, 232
- source, 103

Final build, 234

Fish, dangers of, 34

Flexibility, 46

Formal methods, 220, 221

Four Posts Puzzle, 213

Fowler, Martin, xxiii, 186, 273

Free Software Foundation, see GNU Project

Frog, boiled, see Boiled frog

Function

- accessor, 31
- Law of Demeter for ~s, 140
- similar, 41

G

Gamma, Erich, 194

Garbage collection, 134

Gardening metaphor, 184

Gehrke, Peter, xxv

Glass, Robert, 221, 236

Global variables, 40, 130, 154

Glossary, project, 210

GNU Project, 274

- C/C++ compiler, 267
- General Public License (GPL), 80
- GNU Image Manipulation Program (GIMP), 274
- SmallEiffel, 267

"Good-enough software", see Software, quality

Gordian knot, 212

Goto, 127

GUI system

- vs. command shell, 78
- interface, 78
- testing, 244

Guru, 17, 198

H

Hash, secure, 74

Header file, 29

"Heisenbug", 124, 289

Helicopter, 34n

Hopper, Grace, 8n, 90

"Hot-key" sequence, 196

HTTP Web server, 196
 Human factors, 241
 Humpty Dumpty, xdl, 165
 Hungarian notation, 249
 Hungry consumer model, 153
 Hypertext document, 210

I

iContract, 110, 114, 268
 IDE, *see* Integrated Development Environment
 IEEE Computer Society, 262
 IEEE Computer, 262
 IEEE Software, 263
 Imperative language, 60
 Implementation
 accidents, 173
 coding, 173
 specifications, 219
 Imposed duplication, 28
 Inadvertent duplication, 30
 Indentation, automatic, 85
 Independence, *see* Orthogonality
 Infrastructure, 37
 Inheritance, 111
 assertions, 113
 fan-in/fan-out, 242
 Inner tennis, 215
 Inspection, code, *see* Code reviews
 Insure++, 136
 Integrated circuit, 189n
 Integrated Development Environment (IDE), 72, 232
 Integration platform, 50
 Integration testing, 239
 Interface
 blackboard system, 168
 C/Object Pascal, 101
 concurrency, 155
 error handler, 128
 GUI, 78
 prototyping, 55
 user, 203
 Invariant, 110, 113, 155
 loop, 116
 semantic, 116, 135
 ISO9660 format, 233n
 Iterative development, 69

J

Jacobson, Ivar, 204
 Jargon, xdl, 210
 Jargon file, 273
 Java, 46, 267
 code generation, 232
 DBC, 114
 Enterprise Java Beans, 39, 147
 error messages, 115
 exceptions, 121
 iContract, 110, 114, 268
 javaCC, 59, 269
 JavaDoc, 248, 251
 JavaSpaces, 166, 273
 JUnit, 195
 multithreaded programming, 154
 property access, 100
 property files, 145
 resource balancing, 134
 RMI, 128
 string parser, 156
 tree view, 161
 unit tests, 193
 and Windows shells, 81
 JavaDoc, *see* Java

K

K Desktop Environment, 273
 Kalzen, xdl, 14
 see also Knowledge portfolio
 Kernighan, Brian, 99
 Keybinding, 82
 Kirk, James T., 26
 Knowledge
 producers and consumers, 166
 Knowledge portfolio, 12
 building, 13
 critical thinking, 16
 learning and reading, 14
 researching, 15
 Knuth, Donald, 183, 248
 Korn, David, 81
 Kramer, Reto, xxiv
 Kruchten, Phillipe, 227n

L

Lakos, John, xxiv, 9, 142, 265
 Lame excuses, 3

Language, programming
 conversions, 103, 105
 DBC, 114
 domain, 57
 duplication in, 29
 learning, 14
 prototypes, 55
 scripting, 55, 145
 specification, 58, 62
 text manipulation, 99
 see also Mini-language
Large-Scale C++ Software Design, 142, 265
 L^AT_EX system, 103
 Law of Demeter, 140
 Lawns, care of, xxi
 Layered design, 37
 Layered system, see Modular system
 "lazy" code, 111
 Lex and Yacc, 59
 Librarian, see Project Librarian
 Library code, 39
 Linda model, 167
 Linear algorithms, 177
 Linux, 15, 254, 265
 Liskov Substitution Principle, 111
 Listening, 21
 Literate programming, 248
 Logging, 39, 196
 see also Tracing
 Lookup table, 104
 Loop
 nested, 180
 simple, 180
 Loop invariant, 116

M

Macro, 78, 86
 assertions, 122
 documentation, 252
 error handling, 121
 Maintenance, 26
 imperative languages, 61
 Makefile, 232
 recursive, 233
 Managing expectations, 256
 Mark-up language, 254
 Martin, Robert C., 273

McCabe Cyclomatic Complexity Metric, 242
 Member variables, see Accessor functions
 Memory allocation, 135
 Metadata, 144, 203
 business logic, 146
 configuration, 147
 controlling transactions, 39
 decoupled code, 145
 and formal methods, 221
 in plain text, 74
 Metric, 242
 Meyer, Bertrand, 31n, 109, 184, 264
 Meyer, Scott, 265
 Microsoft Visual C++, 198
 Microsoft Windows, 46
 Mini-language, 59
 data language, 60
 embedded, 62
 imperative, 60
 parsing, 62
 stand-alone, 62
 Mixing board, 205
 MKS Source Integrity, 271
 Model, 160
 calculations, 67
 components and parameters, 66
 and estimating, 66
 executable documents, 251
 view, 162
 Model-view-controller (MVC), 38, 160
 Modular system, 37
 coding, 138
 prototyping, 55
 resource allocation, 135
 reversibility, 45
 testing, 41, 190, 244
More Effective C++, 265
 Mozilla, 273
 Multithreaded programming, 154
 MVC, see Model-view-controller
The Mythical Man Month, 264

N

Name, variable, 249
 Nana, 114, 268
 Nest allocations, 131
 Nested loop, 180

Netscape, 145, 273
 Newsgroup, 15, 17, 33
 Nonorthogonal system, 34
 Normalize, 30
 Novobilski, Andrew J., 189n

O

O() notation, 178, 181
 Object
 coupling, 140n
 destruction, 133, 134
 persistence, 39
 publish/subscribe protocol, 158
 singleton, 41
 valid/invalid state, 154
 viewer, 163
 Object Management Group (OMG), 270
 Object Pascal, 29
 C interface, 101
Object-Oriented Programming, 189n
Object-Oriented Software Construction, 264
 Obsolescence, 74
 OLTP, *see* On-Line Transaction Processing system
 OMG, *see* Object Management Group
 On-Line Transaction Processing system (OLTP), 152
 Options, providing, 3
 Ordering, *see* Workflow
 Orthogonality, 34
 coding, 34, 36, 40
 design, 37
 documentation, 42
 DRY principle, 42
 nonorthogonal system, 34
 productivity, 35
 project teams, 36, 227
 testing, 41
 toolkits & libraries, 39
 see also Modular system
 Over embellishment, 11

P

Pain management, 185
 paint() method, 173
 Painting, 11
 Papua New Guinea, 16

Parallel programming, 150
 Parrots, killer, *see* Branding
 Parsing, 59
 code generators, 105
 log messages, 196
 mini-language, 62
 strings, 155
 Partitioning, 168
 Pascal, 29
 Passive code generator, 103
 Performance testing, 241
 Perl, 55, 62, 99
 C/Object Pascal interface, 101
 database schema generation, 100
 home page, 267
 Java property access, 100
 power tools, 270
 test data generation, 100
 testing, 197
 and typesetting, 100
 Unix utilities in, 81
 web documentation, 101
Perl Journal, 263
 Persistence, 39, 45
 Petzold, Charles, 265
 Pike, Rob, 99
 Pilot
 landing, handling, etc., 217
 who ate fish, 34
 Plain text, 73
 vs. binary format, 73
 drawbacks, 74
 executable documents, 251
 leverage, 75
 obsolescence, 74
 and easier testing, 76
 Unix, 76
 Polymorphism, 111
 Post-it note, 53, 55
 Powerbuilder, 55
The Practice of Programming, 99
 Pragmatic programmer
 characteristics, xviii
 e-mail address, xxiii
 Web site, xxiii
 Pre- and postcondition, 110, 113, 114
 Predicate logic, 110
 Preprocessor, 114
 Presentation, 20

Problem domain, 58, 66
 metadata, 146
 Problem solving, 213
 checklist for, 214
 Productivity, 10, 35
 Programming by coincidence, 173
 Programming staff
 expense of, 237
Programming Windows, 265
 Project
 glossary, 210
 "heads", 228
 saboteur, 244
 schedules, 68
 see also Automation:
 Team, project
 Project librarian, 33, 226
 Prototyping, 53, 216
 architecture, 55
 disposable code, 56
 kinds of, 54
 and programming languages, 55
 and tracer code, 51
 using, 54
 Publish/subscribe protocol, 158
 Pugh, Greg, 95n
 Purify, 136
 PVCS Configuration Management, 271
 Python, 55, 99, 267

Q

Quality
 control, 9
 requirements, 11
 teams, 225
 Quarry worker's creed, xx
 Quicksort algorithm, 180

R

Rational Unified Process, 227n
 Raymond, Eric S., 273
 RCS, see Revision Control System
 Real-world data, 243
 Refactoring, 5, 185
 automatic, 187
 and design, 186
 testing, 187
 time constraints, 185

Refactoring browser, 187, 268
 Refinement, excessive, 11
 Regression, 76, 197, 232, 242
 Relationship
 has-a, 304
 kind-of, 111, 304
 Releases, and SCCS, 87
 Remote Method Invocation (RMI), 128
 exception handling, 39
 Remote procedure call (RPC), 29, 39
 Repository, 87
 Requirement, 11, 202
 business problem, 203
 changing, 26
 creep, 209
 DBC, 110
 distribution, 211
 documenting, 204
 in domain language, 58
 expressing as invariant, 116
 formal methods, 220
 glossary, 210
 over specifying, 208
 and policy, 203
 usability testing, 241
 user interface, 203
 Researching, 15
 Resource balancing, 129
 C++ exceptions, 132
 checking, 135
 coupled code, 130
 dynamic data structures, 135
 encapsulation in class, 132
 Java, 134
 nest allocations, 131
 Response set, 141, 242
 Responsibility, 2, 250, 258
 Reuse, 33, 36
 Reversibility, 44
 flexible architecture, 46
 Revision Control System (RCS), 250, 271
 Risk management, 13
 orthogonality, 36
 RMI, see Remote Method Invocation
 Rock-n-roll, 47
 RPC, see Remote procedure call
 Rubber ducking, 3, 95
 Rules engine, 169

S

Saboteur, 244
 Samba, 272
 Sample programs, *see* Example code
 Sather, 114, 268
 SCCS, *see* Source code control system
 Schedule, project, 68
 Schrödinger, Erwin (and his cat), 47
 Scope, requirement, 209
 Screen scraping, 61
 Scripting language, 55, 145
 Secure hash, 74
 sed, 99
 Sedgewick, Robert, 183
 Self-contained components, *see*
 Orthogonality; Cohesion
 Semantic invariant, 116, 135
 sendmail program, 60
 Sequence diagram, 158
 Server code, 196
 Services, design using, 154
 Shell, command, 77
 vs. GUI, 78
 see also Command shell
 "Shy code", 40
 Side effect, 124
 SIGPLAN, 263
 Simple loop, 180
 Singleton object, 41
 Slashdot, 265
 SmallEiffel, 267
 Smalltalk, 46, 186, 187, 268, 272
 Software
 development technologies, 221
 quality, 9
 requirements, 11
 Software bus, 159
 "Software Construction", 184
Software Development Magazine, 263
 Software IC, 189n
 "Software rot", 4
 Solaris, 76
 Source code
 cat eating, 3
 documentation, *see* Comments
 downloading, *see* Example code
 duplication in, 29
 generating, 103
 reviews, *see* Code reviews

Source code control system (SCCS), 86
 Aegis, 246
 builds using, 88
 CVS, 271
 development tree, 87
 plain text and, 76
 RCS, 250, 271
 repository, 87
 tools, 271
 Specialization, 221
 Specification, 58
 implementation, 219
 language, 62
 as security blanket, 219
 writing, 218
 Spy cells, 138
 Squeak, 268
 Stand-alone mini-language, 62
 "Start-up fatigue", 7
 Starting a project
 problem solving, 212
 prototyping, 216
 specifications, 217
 see also Requirement
 Stevens, W. Richard, 264
 Stone soup, 7
 vs. broken windows, 9
 Stone-cutter's creed, xx
 String parser, 155
 Stroop effect, 249
 strtok routine, 155
 Structured walkthroughs, *see* Code
 reviews
 Style sheet, 20, 254
 Style, communication, 20
 Subclass, 112
 Sublinear algorithm, 177
 Supplier, *see* Vendor
*Surviving Object-Oriented Projects: A
 Manager's Guide*, 264
 SWIG, 55, 270
 Synchronization bar, 151
 Syntax highlighting, 84
 Synthetic data, 243

T

T Spaces, 166, 269
 TAM, *see* Test Access Mechanism
 Tcl, 55, 99, 269

Team, project, 36, 224
 automation, 229
 avoiding duplication, 32
 code review, 236
 communication, 225
 duplication, 226
 functionality, 227
 organization, 227
 pragmatism in, xx
 quality, 225
 tool builders, 229

Technical writer, 252

Template, use case, 205

Temporal coupling, 150

Test Access Mechanism (TAM), 189

Test harness, 194

Testing
 automated, 238
 from specification, 29
 bug fixing, 247
 coverage analysis, 245
 and culture, 197
 debugging, 92, 196
 design/methodology, 242
 effectiveness, 244
 estimates, 182
 frequency, 246
 GUI systems, 244
 integration, 239
 orthogonality, 36, 41
 performance, 241
 role of plain text, 76
 refactoring, 187
 regression, 76, 197, 232, 242
 resource exhaustion, 240
 saboteur, 244
 test data, 100, 243
 usability, 241
 validation and verification, 239
 see also Unit testing

Text manipulation language, 99

TOM programming language, 268

Toolkits, 39

Tools, adaptable, 205

Tracer code, 49
 advantages of, 50
 and prototyping, 51

Tracing, 94
 see also Logging

Trade paper, 263

Trade-offs, 249

Transactions, EJB, 39

Tree widget, 161

troff system, 103

Tuple space, 167

U

UML, see Unified modeling language (UML)

UNDO key, 86

Unified modeling language (UML)
 activity diagram, 150
 sequence diagram, 158
 use case diagram, 208

Uniform Access Principle, 31n

Unit testing, 190
 DBC, 190
 modules, 239
 test harness, 194
 test window, 196
 writing tests, 193

Unix, 46, 76
 Application Default files, 145
 books, 264
 Cygwin, 270
 DOS tools, 270
 Samba, 272
 UWIN, 81, 270

Unix Network Programming, 264

Usability testing, 241

Use case, 204
 diagrams, 206

Usenet newsgroup, 15, 17, 33

User
 expectations, 256
 groups, 18
 interface, 203
 requirements, 10

UWIN, 81, 270

V

Variable
 corrupt, 95
 global, 130, 154
 name, 249

Vendor
 libraries, 39
 reducing reliance on, 36, 39, 46

vi editor, 266

View

debugging, 164

executable documents, 251

Java tree view, 161

model-view-controller, 160, 162

model-viewer network, 162

vim editor, 266

Visual Basic, 55

Visual C++, 198

Visual SourceSafe, 271

VisualWorks, 268

W

Walkthroughs, *see* Code reviews

Warnings, compilation, 92

Web documentation, 101, 210, 253

automatic generation, 235

news and information, 265

Web server, 196

Web site, pragmatic programmer, xxlii

What You See Is What You Get

(WYSIWYG), 78

WikiWikiWeb, 265

Win32 System Services, 265

Windows, 46

"at" command, 231

books, 265

Cygwin, 80

metadata, 145

notepad, 84

Unix utilities, 80, 81

UWIN, 81

WinZip, 272

WISDOM acrostic, 20

Wizard, 198

Word processor, 252, 254

Workflow, 150

blackboard system, 169

content-driven, 234

Wrapper, 132, 133, 135, 141

Writing, 18

see also Documentation

www.pragmaticprogrammer.com, xxlii

WYSIWYG, *see* What You See Is What

You Get

X

XEmacs editor, 266

Xerox Parc, 39

XSL, *see* eXtensible Style Language

xUnit, 194, 269

Y

yacc, 59

Yourdon, Edward, 10, 35

Y2K problem, 32, 208

Z

Z shell, 272

注重实效的程序员之快速参考指南

The Pragmatic Programmer Quick Reference Guide

本参考指南汇总了书中的提示与检查清单

要获得关于 THE PRAGMATIC PROGRAMMERS LLC 的更多信息、各个例子的源码、Web 资源的最新链接、以及在线文献目录, 请访问 www.pragmaticprogrammer.com

1. 关心你的技艺 xix
Care About Your Craft
如果你不在乎能否漂亮地开发出软件, 你又为何要耗费生命去开发软件呢?
2. 思考! 你的工作 xix
Think! About Your Work
关掉自动驾驶仪, 接管操作 不断地批评和评估你的工作
3. 提供各种选择, 不要找蹩脚的借口 3
Provide Options, Don't Make Lame Excuses
要提供各种选择, 而不是找借口 不要说事情做不到; 说明能够做什么
4. 不要容忍破窗户 5
Don't Live with Broken Windows
当你看到糟糕的设计、错误的决策和糟糕的代码时, 修正它们
5. 做变化的催化剂 8
Be a Catalyst for Change
你不能强迫人们改变 相反, 要向他们展示未来可能会怎样, 并帮助他们参与对未来的创造
6. 记住大图景 8
Remember the Big Picture
不要太过专注于细节, 以致忘了查看你周围正在发生什么
7. 使质量成为需求问题 11
Make Quality a Requirements Issue
让你的用户参与确定项目真正的质量需求。

-
8. 定期为你的知识资产投资 14
Invest Regularly in Your Knowledge Portfolio
让学习成为习惯
 9. 批判地分析你读到的和听到的 16
Critically Analyze What You Read and Hear
不要被供应商、媒体炒作、或教条左右 要依照你自己的看法和你的项目的情况去对信息进行分析
 10. 你说什么和你怎么说同样重要 21
It's Both What You Say and the Way You Say It
如果你不能有效地向他人传达你的了不起的想法，这些想法就毫无用处
 11. 不要重复你自己 27
DRY – Don't Repeat Yourself
系统中的每一项知识都必须具有单一、无歧义、权威的表示
 12. 让复用变得容易 33
Make It Easy to Reuse
如果复用很容易，人们就会去复用 创建一个支持复用的环境
 13. 消除无关事物之间的影响 35
Eliminate Effects Between Unrelated Things
设计自足、独立、并具有单一、良好定义的目的的组件
 14. 不存在最终决策 46
There Are No Final Decisions
没有决策是浇筑在石头上的 相反，要把每项决策都视为是写在沙滩上的，并为变化做好计划
 15. 用曳光弹找到目标 49
Use Tracer Bullets to Find the Target
曳光弹能通过试验各种事物并检查它们离目标有多远来让你追踪目标
 16. 为了学习而制作原型 54
Prototype to Learn
原型制作是一种学习经验 其价值并不在于所产生的代码，而在于所学到的经验教训

-
- | | |
|--|----|
| 17. 靠近问题领域编程 | 58 |
| Program Close to the Problem domain | |
| 用你的用户的语言进行设计和编码 | |
| 18. 估算，以避免发生意外 | 64 |
| Estimate to Avoid Surprises | |
| 在着手之前先进行估算——你将提前发现潜在的问题 | |
| 19. 通过代码对进度表进行迭代 | 69 |
| Iterate the Schedule with the Code | |
| 用你在进行实现时获得的经验提炼项目的时间标度 | |
| 20. 用纯文本保存知识 | 74 |
| Keep Knowledge in Plain Text | |
| 纯文本不会过时——它能够帮助你有效利用你的工作，并简化调试和测试。 | |
| 21. 利用命令 shell 的力量 | 80 |
| Use the Power of Command Shells | |
| 当图形用户界面无能为力时使用 shell。 | |
| 22. 用好一种编辑器 | 82 |
| Use a Single Editor Well | |
| 编辑器应该是你的手的延伸；确保你的编辑器是可配置、可扩展和可编程的 | |
| 23. 总是使用源码控制 | 88 |
| Always Use Source Code Control | |
| 源码控制是你的工作的时间机器——你能够回到过去。 | |
| 24. 要修正问题，而不是发出指责 | 91 |
| Fix the Problem, Not the Blame | |
| bug 是你的过错还是别人的过错，并不是真的很有关系——它仍然是你的问题，它仍然需要修正 | |
| 25. 调试时不要恐慌 | 91 |
| Don't Panic When Debugging | |
| 做一次深呼吸，思考什么可能是 bug 的原因 | |

-
- | | |
|--|-----|
| 26. “Select” 没有问题 | 96 |
| “Select” Isn’t Broken | |
| 在 OS 或编译器、甚或是第三方产品或库中很少发现 bug，bug 很可能在应用中 | |
| 27. 不要假定，要证明 | 97 |
| Don’t Assume It – Prove It | |
| 在实际环境中——使用真正的数据和边界条件——证明你的假定 | |
| 28. 学习一种文本操纵语言 | 100 |
| Learn a Text Manipulation Language | |
| 你用每天的很大一部分时间处理文本，为什么不让计算机替你完成部分工作呢？ | |
| 29. 编写能编写代码的代码 | 103 |
| Write Code That Writes Code | |
| 代码生成器能提高你的生产率，并有助于避免重复 | |
| 30. 你不可能写出完美的软件 | 107 |
| You Can’t Write Perfect Software | |
| 软件不可能完美。保护你的代码和用户，使它（他）们免于能够预见的错误 | |
| 31. 通过合约进行设计 | 111 |
| Design with Contracts | |
| 使用合约建立文档，并检验代码所做的事情正好是它声明要做的。 | |
| 32. 早崩溃 | 120 |
| Crash Early | |
| 死程序造成的危害通常比有问题的程序要小得多 | |
| 33. 用断言避免不可能发生的事情 | 122 |
| Use Assertions to Prevent the Impossible | |
| 断言验证你的各种假定。在一个不确定的世界里，用断言保护你的代码。 | |
| 34. 将异常用于异常的问题 | 127 |
| Use Exceptions for Exceptional Problems | |
| 异常可能会遭受经典的意大利面条式代码的所有可读性和可维护性问题的折磨。将异常保留给异常的事物 | |

-
- | | |
|--|-----|
| 35. 要有始有终 | 129 |
| Finish What You Start | |
| 只要可能，分配某资源的例程或对象也应该负责解除其分配 | |
| 36. 使模块之间的耦合减至最少 | 140 |
| Minimize Coupling Between Modules | |
| 通过编写“羞怯的”代码并应用得墨忒耳法则来避免耦合 | |
| 37. 要配置，不要集成 | 144 |
| Configure, Don't Integrate | |
| 要将应用的各种技术选择实现为配置选项，而不是通过集成或工程方法实现 | |
| 38. 将抽象放进代码，细节放进元数据 | 145 |
| Put Abstractions in Code, Details in Metadata | |
| 为一般情况编程，将细节放在被编译的代码库之外 | |
| 39. 分析 workflows，以改善并发性 | 151 |
| Analyze Workflow to Improve Concurrency | |
| 利用你的用户的工作流中的并发性 | |
| 40. 用服务进行设计 | 154 |
| Design Using Services | |
| 根据服务——独立的、在良好定义、一致的接口之后的并发对象——进行设计 | |
| 41. 总是为并发进行设计 | 156 |
| Always Design for Concurrency | |
| 容许并发，你将会设计出更整洁、具有更少假定的接口。 | |
| 42. 使视图与模型分离 | 161 |
| Separate Views from Models | |
| 要根据模型和视图设计你的应用，从而以低廉的代码获取灵活性 | |
| 43. 用黑板协调 workflow | 169 |
| Use Blackboards to Coordinate Workflow | |
| 用黑板协调完全不同的事实和因素，同时又使各参与方保持独立和隔离。 | |

-
- 44. 不要靠巧合编程** 175
Don't Program by Coincidence
只依靠可靠的事物 注意偶发的复杂性，不要把幸运的巧合与有目的的计划混为一谈
- 45. 估算你的算法的阶** 181
Estimate the Order of Your Algorithms
在你编写代码之前，先大致估算事情需要多长时间
- 46. 测试你的估算** 182
Test Your Estimates
对算法的数学分析并不会告诉你每一件事情 在你的代码的目标环境中测定它的速度
- 47. 早重构，常重构** 186
Refactor Early, Refactor Often
就和你会在花园里除草、并重新布置一样，在需要时对代码进行重写、重做和重新架构 要铲除问题的根源
- 48. 为测试而设计** 192
Design to Test
在你还没有编写代码时就开始思考测试问题
- 49. 测试你的软件，否则你的用户就得测试** 197
Test Your Software, or Your Users Will
无情地测试 不要让你的用户为你查找 bug
- 50. 不要使用你不理解的向导代码** 199
Don't Use Wizard Code You Don't Understand
向导可以生成大量代码 在你把它们合并进你的项目之前，确保你理解全部这些代码
- 51. 不要搜集需求——挖掘它们** 202
Don't Gather Requirements – Dig for Them
需求很少存在于表面上 它们深深地埋藏在层层假定、误解和政治手段的下面

-
52. 与用户一同工作，以像用户一样思考 204
Work with a User to Think Like a User
要了解系统实际上将如何被使用，这是最好的方法
53. 抽象比细节活得更长久 209
Abstractions Live Longer than Details
“投资”于抽象，而不是实现 抽象能在来自不同的实现和新技术的变化的“攻击”之下存活下去
54. 使用项目词汇表 210
Use a Project Glossary
创建并维护项目中使用的专用术语和词汇的单一信息源
55. 不要在盒子外面思考——要找到盒子 213
Don't Think Outside the Box – Find the Box
在遇到不可能解决的问题时，要确定真正的约束 问问你自己：“它必须以这种方式完成吗？它真的必须完成吗？”
56. 等你准备好再开始 215
Start When You're Ready
你的一生都在积累经验，不要忽视反复出现的疑虑
57. 对有些事情“做”胜于“描述” 218
Some Things Are Better Done than Described
不要掉进规范的螺旋——在某个时刻，你需要开始编码。
58. 不要做形式方法的奴隶 220
Don't Be a Slave to Formal Methods
如果你没有把某项技术放进你的开发实践和能力的语境中，不要盲目地采用它。
59. 昂贵的工具不一定能制作出更好的设计 222
Costly Tools Don't Produce Better Designs
小心供应商的炒作，行业教条，以及价格标签的诱惑，要根据工具的价值判断它们。
60. 围绕功能组织团队 227
Organize Teams Around Functionality
不要把设计师与编码员分开，也不要把测试员与数据建模员分开 按照你构建代码

的方式构建团队。

- | | |
|---|-----|
| 61. 不要使用手工流程 | 231 |
| Don't Use Manual Procedures | |
| shell 脚本或批文件会一次次地以同一顺序执行同样的指令。 | |
| 62. 早测试，常测试，自动测试。 | 237 |
| Test Early. Test Often. Test Automatically | |
| 与呆在书架上的测试计划相比，每次构建时运行的测试要有效得多 | |
| 63. 要到通过全部测试，编码才算完成 | 238 |
| Coding Ain't Done 'Til All the Tests Run | |
| 就是这样 | |
| 64. 通过“蓄意破坏”测试你的测试 | 244 |
| Use Saboteurs to Test Your Testing | |
| 在单独的软件副本上故意引入 bug，以检验测试能够抓住它们 | |
| 65. 测试状态覆盖，而不是代码覆盖 | 245 |
| Test State Coverage, Not Code Coverage | |
| 确定并测试重要的程序状态。只是测试代码行是不够的。 | |
| 66. 一个 bug 只抓一次 | 247 |
| Find Bugs Once | |
| 一旦测试员找到一个 bug，这应该是测试员最后一次找到它。此后自动测试应该对其进行检查 | |
| 67. 英语就是一种编程语言 | 248 |
| English is Just a Programming Language | |
| 像你编写代码一样编写文档：遵守 <i>DRY</i> 原则、使用元数据、MVC、自动生成，等等 | |
| 68. 把文档建在里面，不要拴在外面 | 248 |
| Build Documentation In, Don't Bolt It On | |
| 与代码分离的文档不太可能被修正和更新。 | |

69. 温和地超出用户的期望 255
 Gently Exceed Your Users' Expectations
 要理解你的用户的期望，然后给他们的东西要多那么一点
70. 在你的作品上签名 258
 Sign Your Work
 过去时代的手艺人为能在他们的作品上签名而自豪，你也应该如此。

检查清单

71. 要学习的语言 17 页
 厌倦了 C、C++ 和 JAVA？试试 CLOS、Dylan、Eiffel、Objective C、Prolog、Smalltalk 或 TOM。它们每一种都有不同的能力和不同的“风味”。用其中的一种或多种语言在家里开发一个小项目

72. WISDOM 离合诗 20 页
- | | |
|---|--------------|
| What do you want them to learn? | 你想让他们学到什么？ |
| What is their interest in what you've got to say? | 他们对你讲的什么感兴趣？ |
| How sophisticated are they? | 他们有多富有经验？ |
| How much detail do they want? | 他们想要多少细节？ |
| Whom do you want to own the information? | 你想要让谁拥有这些信息？ |
| How can you motivate them to listen to you? | 你如何促使他们听你说话？ |

73. 怎样维持正交性 34 页
- 设计独立、良好定义的组件。
 - 使你的代码保持解藕
 - 避免使用全局数据。
 - 重构相似的函数。
74. 应制作原型的事物 53 页
- 架构
 - 已有系统中的新功能

- 外部数据的结构或内容
- 第三方工具或组件
- 性能问题
- 用户界面设计

75. 架构问题

55 页

- 责任是否得到了良好定义?
- 协作是否得到了良好定义?
- 耦合是否得以最小化?
- 你能否确定潜在的重复?
- 接口定义和各项约束是否可接受?
- 模块能否在需要时访问所需数据?

76. 调试检查清单

98 页

- 正在报告的问题是底层 bug 的直接结果，还是只是症状?
- bug 真的在编译器里？在 OS 里？或者是在你的代码里？
- 如果你向同事详细解释这个问题，你会说什么？
- 如果可疑代码通过了单元测试，测试是否足够完整？如果你用该数据运行单元测试，会发生什么？
- 造成这个 bug 的条件是否存在于系统中的其他任何地方？

77. 函数的得墨忒耳法则

141 页

某个对象的方法应该只调用属于以下情形的方法：

- 它自身
- 传入的任何参数
- 它创建的对象
- 组件对象

78. 怎样深思熟虑地编程

172 页

- 总是意识到你在做什么。
- 不要盲目地编程
- 按照计划行事
- 依靠可靠的事物
- 为你的假定建立文档
- 不要只是测试你的代码，还要测试你的假定

- 为你的工作划分优先级
- 不要做历史的奴隶

79. 何时进行重构 185 页

- 你发现了对 *DRY* 原则的违反
- 你发现事物可以更为正交
- 你的知识扩展了
- 需求演变了
- 你需要改善性能

80. 劈开戈尔迪斯结 212 页

在解决不可能解决的问题时，问问你自己：

- 有更容易的方法吗？
- 我是在解决正确的问题吗？
- 这件事情为什么是一个问题？
- 是什么使它如此难以解决？
- 它必须以这种方式完成吗？
- 它真的必须完成吗？

81. 测试的各个方面 237 页

- 单元测试
- 集成测试
- 验证和校验
- 资源耗尽、错误及恢复
- 性能测试
- 可用性测试
- 对测试自身进行测试